CARLETON COLLEGE
QA76.73.C28 Y82 1988
Yuasa, T./Common Lisp drill
Staxs
3 0118 00308 4021

Taiichi Yuasa

Common Lisp Drill

BOOK card + date due slip
on next page

Common Lisp Drill

WITHDRAWN

QA 76.73 .C28 Y82 1988

Yuasa, T.

Common Lisp drill

AP 7'89 BRIAN HAYES 104 MADISON

.C28 Y82 1988

DEMCO

DATE DUE

MAY 1 1989

DEMCO 38-297

Com

Taiichi

Research Ins
Kyoto Unive
Kyoto, Japan

Translat
Richard

ACADEMIC PRESS, INC.
Harcourt Brace Jovanovich, Publishers

Boston San Diego New York
Berkeley London Sydney
Tokyo Toronto

QA
76.73
.C28
Y82
1988

Copyright © 1987 by Taiichi Yuasa.
Originally published in Japanese by Iwanami Shoten, Publishers, Tokyo, 1987.
This English language edition © 1988 by Academic Press, Inc.

All rights reserved.
No part of this publication may be reproduced or
transmitted in any form or by any means, electronic
or mechanical, including photocopy, recording, or
any information storage and retrieval system, without
permission in writing from the publisher.

ACADEMIC PRESS, INC.
1250 Sixth Avenue, San Diego, CA 92101

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24-28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Yuasa, T. (Taiichi)
Common Lisp drill/Taiichi Yuasa: translated by Richard
Weyhrauch and Yasuko Kitajima.
p. cm.
"Originally published in Japanese by Iwanami Shoten, Publishers,
Tokyo, 1987."
Bibliography:
Includes index.
ISBN 0-12-774861-X
1. COMMON LISP (Computer program language) I. Title.
QA76.73.C28Y82 1988 88-6392
005.13'3–dc 19 CIP

88 89 90 91 9 8 7 6 5 4 3 2 1
Printed in the United States of America

Contents

Preface ix

Chapter 1: Basic Constructs 1

1-1 A Common Lisp System 3
1-2 Functions 4
1-3 Variables and Symbols 5
1-4 Lists 6
1-5 Evaluation and Forms 8
Answers to Exercises 9

Chapter 2: Function Definitions 13

2-1 How to Define a Function 15
2-2 Local Variables 15
2-3 Predicates and True-False Values 16
2-4 Elements of Logical Expressions 17
2-5 Conditional Expressions 18
2-6 Multiple-Valued Functions 18
Answers to Exercises 20

Chapter 3: Control Structure 25

3-1 Recursive Call 26
3-2 Block Structure 34
3-3 Program Structure 37
3-4 Macros for Iteration Constructs 37
3-5 `catch` and `throw` 39
Answers to Exercises 42

Chapter 4: Symbols and Packages **53**

4-1 Symbol Names and Print Names 54
4-2 Packages 55
4-3 Internal Symbols and External Symbols 55
4-4 Interning 57
4-5 Predefined Packages 60
4-6 Shadowing 60
4-7 Functions for Making Symbols 61
Answers to Exercises 62

Chapter 5: List Structure **67**

5-1 cons Cells 68
5-2 Destructive List Operations 71
5-3 Equality 74
5-4 Garbage Collection 78
Answers to Exercises 80

Chapter 6: Functions **89**

6-1 Lambda Expressions 90
6-2 Local Functions 93
6-3 Functions for Function Calls 95
6-4 Lexical Closures 97
Answers to Exercises 98

Chapter 7: Macros **103**

7-1 Macros 104
7-2 `setf` 104
7-3 Macro Definitions 106
7-4 Backquote 108
Answers to Exercises 110

Chapter 8: List Processing **115**

8-1 List Processing Functions 117
8-2 Mapping Functions 118
8-3 Alists 123
8-4 Property Lists 127
Answers to Exercises 130

Chapter 9: Declarations **145**

9-1 Dynamic Binding 146
9-2 Special Variables 146
9-3 Special Declaration 154
9-4 Variables and Constants 158
Answers to Exercises 159

Chapter 10: Data Types **167**

10-1 Numbers 170
10-2 Arrays and Vectors 172
10-3 Characters and Strings 174
10-4 Structures 178
Answers to Exercises 181

Chapter 11: Input and Output **189**

11-1 Streams 191
11-2 Outputting Data 192
11-3 Inputting Data 194
11-4 Input and Output of Characters 195
11-5 File System 197
11-6 Input and Output to Strings 200
11-7 Read Macros 202
11-8 Sharp Sign Macro 202
11-9 Formatted Output 203
Answers to Exercises 208

Chapter 12: Programming **221**

12-1 A Program File 222
12-2 Compiler 222
12-3 Module Structure 224
12-4 Features 225
12-5 Miscellaneous Information 225
Answers to Exercises 226

Index **231**

Preface

This book contains Common Lisp exercises based on *Introduction to Common Lisp*, which was originally published in Japanese in 1986 by Iwanami Shoten, Publishers and was translated in 1987 by Academic Press. *Introduction to Common Lisp* was written in cooperation with Mr. Masami Hagiya.

Each of the twelve chapters here corresponds to a chapter of *Introduction to Common Lisp*. Each chapter contains the following four sections:

1. A summary of the chapter
2. A summary of the functions and the variables
3. Exercises
4. Answers to exercises

The summary of the chapter briefly reports what the reader learned in its corresponding chapter of *Introduction to Common Lisp*. If the reader understands the summary, he or she understood most of the content of the chapter. The summaries of the functions and variables are lists of the special forms, macros, functions, variables, etc. that were introduced in the corresponding chapter of *Introduction to Common Lisp*. Please review what each of them does. Since special forms, macros, and functions are shown with their argument types, the reader can refer to this section when he or she actually uses them. Exercises are given for each section in *Introduction to Common Lisp*. There are various levels of exercises; simple ones that ask for the values of expressions and fill in answers for empty parts of interactions and difficult ones which relate to the use of Lisp for applications in artificial intelligence. As in *Introduction to Common Lisp*, we do not make assumptions about a reader's previous knowledge, and we explain

each new notion in the exercise. Exercises asking for the definition of a function contain examples of using such a function so the reader can better understand the intention of the exercise. Also, these examples can be used to test whether the reader's definition works correctly. Answers to all the exercises are given at the end of the chapter. We have added some comments to some of the exercises. Usually the answer given to an exercise is just one example of many possible answers. It may not match the reader's answer. However, it is very useful for beginning programmers to study the programming style of programs written by other people, so the answers should be studied carefully.

We used the following convention in describing forms:

1. "⟪A⟫" means `A` itself appears in the expression. "⟨A⟩" means a form which becomes `A` after evaluation.
2. "`A...A`" means a row of `A`'s. This row can be empty.
3. When `A` and `B` are different, "`AB...AB`" represents a row of alternatating `A` and `B`'s. This row can be empty.
4. "`[A]`" means that `A` can be omitted.
5. Keywords are placed between "{" and "}" when a function takes keyword parameters.

Please note that the summary of the functions uses only the structures introduced in *Introduction to Common Lisp*.

This book is a collection of the problems used as exercises during the last semester of 1986 in the programming class for Seniors in the Department of Science of Kyoto University. I would like to thank Professor Reiji Nakajima, for giving me an opportunity to teach this course, and the students who actually tried these problems and gave me useful comments. This book would not have existed without *Introduction to Common Lisp*. I would like to thank Mr. Masami Hagiya who coauthored *Introduction to Common Lisp* and who had many discussions with me about Common Lisp. Lastly, I would like to thank my wife Keiko and my daughter Makiko who cheered me on while writing this book.

Let's start practicing Common Lisp. I look forward to seeing good Lisp programs from the readers.

February 1987
Taiichi Yuasa

1 Basic Constructs

A Lisp system repeats the following operations:

displays a prompt
reads one form
evaluates the form
displays the results

A form is a Lisp data structure. Some functions accept and evaluate forms. Forms are divided into the following groups:

1. List Forms
   1.1 Special Forms
   1.2 Macro Forms
   1.3 Function Call Forms
2. Variables
3. Forms (data) that evaluate to themselves

While a user cannot make new special forms, he or she can make or change macros and functions freely.

The rules for evaluating each special form is specific to the form. Each one has a different specification. The purpose of a macro form is to change one form into another form before it is evaluated. When a function call form is evaluated, the expressions for its arguments are evaluated first, and

their values are then given to the function. A variable is a box for keeping data and always has a name. When a variable is evaluated, the system returns the contents of the box as its value.

The major data structures of Lisp are symbols, numbers, and lists. A list is either the empty list `nil` or can be constructed using the `cons` function or using the `quote` form (or its abbreviation, the quote, "'", mark). A `quote` form can be used for producing not only lists but also any particular data.

- summary of functions

  (`+` ⟨number⟩ . . . ⟨number⟩)
  (`*` ⟨number⟩ . . . ⟨number⟩)
  (`floor` ⟨number⟩ ⟨number⟩)

  (`setq` ⟪⟨variable⟩⟫ ⟨value⟩ . . . ⟪⟨variable⟩⟫ ⟨value⟩)
  (`psetq` ⟪⟨variable⟩⟫ ⟨value⟩ . . . ⟪⟨variable⟩⟫⟨value⟩)
  (`set` ⟨symbol⟩ ⟨value⟩)
  (`symbol-value` ⟨symbol⟩)

  (`cons` ⟨data⟩⟨data⟩)
  (`list` ⟨data⟩ . . . ⟨data⟩)
  (`length` ⟨list⟩)
  (`car` ⟨list⟩)
  (`cdr` ⟨list⟩)
  (`caar` ⟨list⟩)
  (`cadr` ⟨list⟩)
  (`cdar` ⟨list⟩)
  (`cddr` ⟨list⟩)
  (`caaar` ⟨list⟩)
  (`caadr` ⟨list⟩)
  (`cadar` ⟨list⟩)
  (`caddr` ⟨list⟩)
  (`cdaar` ⟨list⟩)
  (`cdadr` ⟨list⟩)
  (`cddar` ⟨list⟩)
  (`cdddr` ⟨list⟩)
  (`caaaar` ⟨list⟩)
  (`caaadr` ⟨list⟩)
  (`caadar` ⟨list⟩)
  (`caaddr` ⟨list⟩)
  (`cadaar` ⟨list⟩)

(cadadr ⟨list⟩)
(caddar ⟨list⟩)
(cadddr ⟨list⟩)
(cdaaar ⟨list⟩)
(cdaadr ⟨list⟩)
(cdadar ⟨list⟩)
(cdaddr ⟨list⟩)
(cddaar ⟨list⟩)
(cddadr ⟨list⟩)
(cdddar ⟨list⟩)
(cddddr ⟨list⟩)

(quote ⟪data⟫)
(eval ⟨form⟩)
(macroexpand ⟨form⟩)

- summary of constant variables

t
nil
pi

1-1 A Common Lisp System

1.1.1 The following shows the interaction with the operating system when KCL is started on a UNIX machine. Describe which lines are input by a user.

```
% kcl
KCL (kyoto Common Lisp) October 9, 1986

>(+ 1 2)
3

>(* 3 5)
15

>
```

1-2 Functions

1.2.1 What is the value of each of the following expressions:

a. `(+ (* 4 4) 4)`
b. `(* (+ 1 3) 5)`
c. `(+ (+ 2 2) (* (+ 2 2) (* 2 2)))`
d. `(* (floor (+ 1 4) 1) (floor 16 4))`

1.2.2 When there are many nested function expressions in one expression, the value of the whole expression can be easily obtained by replacing each function call expression with its value. For example,

```
(+ (* 3 4) (floor 11 3))
→ (+ 12 (floor 11 3))
→ (+ 12 3)
→ 15
```

We can evaluate the four expressions in Exercise 1.2.1 using this method. Begin by filling in the appropriate numbers in the spaces `[ ]` below.

a. `(+ (* 4 4) 4)`
```
   → (+ [ ] 4)
   → [ ]
```

b. `(* (+ 1 3) 5)`
```
   → (* [ ] 5)
   → [ ]
```

c. `(+ (+ 2 2) (* (+ 2 2) (* 2 2)))`
```
   → (+ 4 (* (+ 2 2) (* 2 2)))
   → (+ 4 (* [ ] (* 2 2)))
   → (+ 4 (* [ ] [ ]))
   → (+ 4 [ ])
   → [ ]
```

d. `(* (floor (+ 1 4) 1) (floor 16 4))`
```
   → (* (floor 5 1) (floor 16 4))
   → (* 5 (floor 16 4))
   → (* 5 [ ])
   → [ ]
```

1.2.3 The system function `max` takes some numbers as arguments and returns the largest one as a value. There should be more than one argument. The system function `min` returns the smallest number among its arguments. Find the value of each of the following expressions:

a. `(max 1 2)`
b. `(min 1 2)`
c. `(max 7 2 3 8)`
d. `(min 7 2 3 8)`
e. `(max (+ 2 3) (* 2 3))`
f. `(* (max 7 2 3 8) (min 7 2 3 8))`
g. `(max (floor 17 5) (min 17 4))`

1-3 Variables and Symbols

1.3.1 Suppose the values of the three variables, `hour`, `minute`, and `second` are 10, 20, and 30, respectively. Find the values of the following expressions:

a. `(* hour 60)`
b. `(* (+ (* hour 60) minute) 60)`
c. `(+ (* (+ (* hour 60) minute) 60) second)`

1.3.2 Fill in [ ] in the following interaction:

```
>(setq x 3)
[ ]

>(+ (* x x) x)
[ ]

>(setq y 5 z 7)
[ ]

>(+ x y z)
[ ]

>(setq x 0)
[ ]

>(+ x y z)
[ ]
```

```
>(setq x (+ y z))
[ ]

>x
[ ]
```

1.3.3 Suppose the value of a variable x is 2 and the value of y is 3. What are the values of x and y when

```
(psetq x y y x)
```

is evaluated. What are the values of x and y when

```
(setq x y y x)
```

is evaluated.

1.3.4 Check what your system does when you try to assign something to a system constant variable. For example, what happens if you type

```
>(setq pi 100)
```

Also, what happens if you type

```
>(set 'pi 100)
```

1-4 Lists

1.4.1 What are the values of the following expressions:

a. `(list 0 0 7)`
b. `(list 1 (quote +) 2 (quote =) 3)`
c. `(list '* 'is 'a 'symbol)`
d. `(list 'he 'has 3 'dogs)`
e. `(list '+ (list '* 2 3) 5)`

1.4.2 Fill in [ ] in the following interaction:

```
>(setq x 'dogs)
[ ]

>(setq x (list 'ugly x))
[ ]

>(setq x (list 3 x))
[ ]

>(setq x (list 'he 'has x))
[ ]
```

1.4.3 Suppose the value of a variable `rgb` is the list `(RED GREEN BLUE)`. What are the values of the following expressions:

a. `(car rgb)`
b. `(cdr rgb)`
c. `(car (cdr rgb))`
d. `(cdr (cdr rgb))`
e. `(car (cdr (cdr rgb)))`
f. `(cdr (cdr (cdr rgb)))`
g. `(car (cdr (cdr (cdr rgb))))`
h. `(cdr (cdr (cdr (cdr rgb))))`

1.4.4 Fill in [ ] with `car` or `cdr`.

a. `(caddr x)` has the same value as `([ ]([ ]([ ] x)))`.
b. `(cadadr y)` has the same value as `([ ]([ ]([ ]([ ] y))))`.
c. `(cddddr z)` has the same value as `([ ]([ ]([ ]([ ] z ))))`.

1.4.5 What are the values of the following expressions when the value of the variable x is `((A 1) (B 2) (C 3))`.

a. `(caddr x)`
b. `(car (cadr x))`
c. `(cadr (cadr x))`
d. `(length x)`
e. `(length (car x))`
f. `(cons '(z 4) x)`
g. `(cons '(w 5) (cons '(z 4) x))`
h. `(cons '(w 5) (cdr x))`
i. `(cons (caddr x) (cons (cadr x) (cons (car x) nil)))`
j. `(list (car x) (cadr x) (caddr x))`

1-5 Evaluation and Forms

1.5.1 What are the values of the following expressions when the value of the variable x is the list `(* 2 3)`:

a. `(car x)`
b. `(eval x)`
c. `(eval 'x)`
d. `(eval (quote x))`
e. `(eval (quote (quote x)))`
f. `(eval (quote 'x))`
g. `(eval ''x)`

1.5.2 Which of the following expressions are **not** forms?

a. `123456789`
b. `(quote (quote quote quote))`
c. `variable`
d. `(0)`
e. `(setq boo 'hay foo 'wood woo 'brick)`
f. `nil`
g. `(incf my-money)`
h. `( )`
i. `(quote (1 2 3 4))`
j. `(+ hop skip jump)`

Also, specify which of the above are special forms, macro forms, and function call forms.

Answers to Exercises

1.1.1 The first "kcl", the form "(+ 1 2)", and the form "(* 3 5)" are the forms input by the user. Everything else is output by the system.

1.2.1 a. 20 b. 20 c. 20 d. 20

1.2.2

a.
```
(+ (* 4 4) 4)
   → (+ [ 16 ] 4)
   → [ 20 ]
```

b.
```
(* (+ 1 3) 5)
   → (* [ 4 ] 5)
   → [ 20 ]
```

c.
```
(+ (+ 2 2) (* (+ 2 2) (* 2 2)))
   → (+ 4 (* (+ 2 2) (* 2 2)))
   → (+ 4 (* [ 4 ] (* 2 2)))
   → (+ 4 (* [ 4 ] [ 4 ]))
   → (+ 4 [ 16 ])
   → [ 20 ]
```

d.
```
(* (floor (+ 1 4) 1) (floor 16 4))
   → (* (floor 5 1) (floor 16 4))
   → (* 5 (floor 16 4))
   → (* 5 [ 4 ])
   → [ 20 ]
```

1.2.3 a. 2 b. 1 c. 8 d. 2 e. 6 f. 16 g. 4

1.3.1 a. 600 b. 37200 c. 37230

1.3.2

```
>(setq x 3)
[ 3 ]

>(+ (* x x) x)
[ 12 ]

>(setq y 5 z 7)
[ 7 ]
```

```
>(+ x y z)
[ 15 ]

>(setq x 0)
[ 0 ]

>(+ x y z)
[ 12 ]

>(setq x (+ y z))
[ 12 ]

>x
[ 12 ]
```

1.3.3 After `(psetq x y y x)` is evaluated, `x` becomes 3 and `y` becomes 2. In other words, this `psetq` expression exchanges the value of `x` and `y`. In `(setq x y y x)`, the value of `x` becomes 3 and then the value of `x` is assigned to `y`. As a result, the value of both `x` and `y` becomes 3.

1.3.4 In KCL, we will get

```
>(setq pi 100)
Error: Cannot assign to the constant PI.
Error signalled by SETQ.

Broken at SETQ.  Type :H for Help.
>>:q

Top level.
>(set 'pi 100)
Error: Cannot assign to the constant PI.
Error signalled by SET.

Broken at SET.  Type :H for Help.
>>:q

Top level.
>
```

Different Lisps may give different responses, but in each case an error should be signalled.

1.4.1

a. (0 0 7)
b. (1 + 2 = 3)
c. (* IS A SYMBOL)
d. (HE HAS 3 DOGS)
e. (+ (* 2 3) 5)

1.4.2

```
>(setq x 'dogs)
[ DOGS ]

>(setq x (list 'ugly x))
[ (UGLY DOGS) ]

>(setq x (list 3 x))
[ (3 (UGLY DOGS)) ]

>(setq x (list 'he 'has x))
[ (HE HAS (3 (UGLY DOGS))) ]
```

1.4.3

a. RED
b. (GREEN BLUE)
c. GREEN
d. (BLUE)
e. BLUE
f. NIL
g. NIL
h. NIL

1.4.4

a. ([car]([cdr]([cdr] x)))
b. ([car]([cdr]([car]([cdr] y))))
c. ([cdr]([cdr]([cdr]([cdr] z))))

1.4.5

a. (C 3)
b. B
c. 2
d. 3
e. 2

f. `((Z 4) (A 1) (B 2) (C 3))`
g. `((W 5) (Z 4) (A 1) (B 2) (C 3))`
h. `((W 5) (B 2) (C 3))`
i. `((C 3) (B 2) (A 1))`
j. `((A 1) (B 2) (C 3))`

1.5.1

a. `*`
b. `6`
c. `(* 2 3)`
d. `(* 2 3)`
e. `X`
f. `X`
g. `X`

In order to evaluate `(car x)`, you first need to determine the value of `x`, which is the list `(* 2 3)`, and then use this value as an argument to `car`. Therefore, the final value is the symbol `*`. In the same way, during the evaluation of `(eval x)`, you must first get the value of `x` and use this value, `(* 2 3)`, as an argument to `eval`. In this case the final value is 6. In the case of `(eval 'x)`, the argument to `eval` is the symbol `x`, and `eval` returns the value of `x`, namely the list `(* 2 3)`, as a value. Since `'x` is an abbreviation of `(quote x)`, `(eval 'x)` is an abbreviation of `(eval (quote x))`. Therefore, the value of `(eval (quote x))` will also be `(* 2 3)`. In `(eval (quote (quote x)))`, the value of `(quote (quote x))`, namely, `(quote x)` will be given to `eval`. Since the value of `(quote x)` is a symbol `x`, the value of the `eval` will be the symbol `x`. Both `(eval (quote 'x))` and `(eval ''x)` are abbreviations of the form `(eval (quote (quote x)))`, so their value will be the same as the value of `(eval (quote (quote x)))`.

These examples may seem complicated. Please keep the following four things in mind and try to understand them:

1. `eval` is a function;
2. Arguments to functions always get evaluated before they are given to a function;
3. '⟪data⟫ is an abbreviation of the form `(quote` ⟪data⟫`)`;
4. The value of `(quote` ⟪data⟫`)` is ⟪data⟫ itself.

1.5.2 Example d is not a form; b, e, and i are special forms; g is a macro form; j is a function call form; a is a number and is a form whose value is itself; and c, f, and h are symbols that are treated as variables.

2 Function Definitions

In order to define a function, we use the **defun** form in which we specify the function name, its parameter list, and its body. A parameter is a local variable which can only be referred to inside the body of the **defun** form. Arguments are passed to a function by binding these parameters. We use the **let** form and the **let*** form when we want to use local variables instead of parameters.

In Lisp, the symbol **nil** is used to mean the logical value "false." Any value except **nil** is considered to be "true." Conditional expressions containing **if**, **when**, or **cond** use this method in order to decide whether their conditions are true or false. Conditions are made up of the predicates, **not**, **and**, **or**, etc.

Although functions and macros can return multiple values, you should generally think of them as returning only one value. On the other hand, functions such as **values**, macros such as **multiple-value-bind**, and some special forms make use of the ability of functions to return multiple values.

- summary of functions

```
(defun ⟪function name⟫  ⟪parameter list⟫
    ⟪expression⟫ . . . ⟪expression⟫ )

(let  (( ⟪variable⟫ ⟨initial value⟩)
       . . .
       ( ⟪variable⟫ ⟨initial value⟩))
  ⟪expression⟫ . . . ⟪expression⟫ )
```

```
(let* (( ⟪variable⟫ ⟨initial value⟩)
       ...
       ( ⟪variable⟫ ⟨initial value⟩))
  ⟪expression⟫ ... ⟪expression⟫ )

(progn ⟪expression⟫ ... ⟪expression⟫ )
(prog1 ⟪expression⟫ ⟪expression⟫ ... ⟪expression⟫ )

(null ⟨data⟩)
(not ⟨data⟩)
(and ⟪condition⟫ ... ⟪condition⟫ )
(or ⟪condition⟫ ... ⟪condition⟫ )

(< ⟨number⟩ ⟨number⟩ ... ⟨number⟩)
(> ⟨number⟩ ⟨number⟩ ... ⟨number⟩)
(= ⟨number⟩ ⟨number⟩ ... ⟨number⟩)
(/= ⟨number⟩ ⟨number⟩ ... ⟨number⟩)
(<= ⟨number⟩ ⟨number⟩ ... ⟨number⟩)
(>= ⟨number⟩ ⟨number⟩ ... ⟨number⟩)

(numberp ⟨data⟩)
(plusp ⟨number⟩)
(zerop ⟨number⟩)
(minusp ⟨number⟩)
(evenp ⟨number⟩)
(oddp ⟨number⟩)

(if ⟪condition⟫ ⟪expression⟫ [ ⟪expression⟫ ])
(when ⟪condition⟫ ⟪expression⟫ ... ⟪expression⟫ )
(unless ⟪condition⟫ ⟪expression⟫ ... ⟪expression⟫ )
(cond ( ⟪condition⟫ ⟪expression⟫ ... ⟪expression⟫ )
      ...
      ( ⟪condition⟫ ⟪expression⟫ ... ⟪expression⟫ ))

(case ⟨key value⟩
  ( ⟪key list⟫ ⟪expression⟫ ... ⟪expression⟫ )
  ...
  ( ⟪key list⟫ ⟪expression⟫ ... ⟪expression⟫ ))

(values ⟨data⟩ ... ⟨data⟩)
(values-list ⟨list⟩)
(multiple-value-setq
   ( ⟪variable⟫ ... ⟪variable⟫ ) ⟪expression⟫ )
```

```
(multiple-value-bind
   ( ⟪variable⟫ ... ⟪variable⟫ ) ⟪expression⟫
        ⟪expression⟫ ... ⟪expression⟫ )
(multiple-value-list ⟪expression⟫ )
(multiple-value-prog1 ⟪expression⟫
   ⟪expression⟫ ... ⟪expression⟫ )
(multiple-value-call ⟨function⟩ ⟪expression⟫ ... ⟪expression⟫ )
```

2-1 How to Define a Function

2.1.1 System functions such as `cadr`, `cdddr`, and `cadadr` which were introduced in Chapter 1 of *Introduction to Common Lisp* can be defined by combining `car` and `cdr`. For this example, define, using `car` and `cdr`, the functions `my-cadr`, `my-cdddr`, and `my-cadadr` which have the same value as `cadr`, `cdddr`, and `cadadr`, respectively.

2.1.2 Define a function `poly-2-3-4` which evaluates the second degree polynomial

$$2x^2 + 3x + 4$$

Suppose the function takes the value of `x` as a parameter. For example,

```
>(poly-2-3-4 10)
234

>(poly-2-3-4 0)
4
```

2.1.3 Define a function `area` which takes radius of a circle and returns the area of the circle as a value. Use the system constant `pi` as the ratio of the radius to the circumference.

2-2 Local Variables

2.2.1 Explain what happens in the following function when it is given two numbers as arguments:

```
(defun guess-what (x y)
  (let ((a (max x y))
        (b (min x y)))
    (- a b)))
```

Here - is the system function for subtraction.

2.2.2 Describe the effect of the following expression:

```
(let ((v 3))
  (list v
        (let ((v (* v v)))
          (+ v v))
        v
        (let ((v (+ v v)))
          (* v v))
        v))
```

2-3 Predicates and True-False Values

2.3.1 Suppose a function `foo` is defined as follows:

```
(defun foo (x y)
  (list (< x y)
        (<= x y)
        (= x y)
        (>= x y)
        (> x y)))
```

What are the values of the following expressions?

a. `(foo 1 2)`
b. `(foo 2 2)`
c. `(foo 3 2)`

2.3.2 Suppose a function `foo` is defined as follows:

```
(defun foo (x)
  (list (plusp x)
        (zerop x)
        (minusp x)
        (evenp x)
        (oddp x)))
```

What are the values of the following expressions?

a. `(foo -2)`
b. `(foo -1)`
c. `(foo 0)`
d. `(foo 1)`
e. `(foo 2)`

2.3.3 A predicate `plusp` can be defined as follows using `>`:

```
(defun plusp (x) (> x 0))
```

Define predicates `minusp` and `zerop` using `<` and `=`, respectively.

2.3.4 The system function `mod` is used for calculating the remainder after division.

`(mod` ⟨the number to be divided⟩ ⟨the number to divide by⟩`)`

returns the remainder after ⟨the number to be divided⟩ is divided by ⟨the number to divide by⟩. Define, using `mod`, the function `my-evenp` which has the same value as the function `evenp`.

2.3.5 Define a function `adenoy` which takes an integer, divides it by 2 if it is an even number, and subtracts 1 from it if it is an odd number.

2-4 Elements of Logical Expressions

2.4.1 Define a function `my-oddp`, using a function `mod` (see Exercise 2.3.4), which has the same value as `oddp`.

2.4.2 Write an expression, using an `if` expression, which has the same value as

```
(and (numberp x) (>= x 0) (<= x 100))
```

2.4.3 Write an expression, using `if` expression and `let` expression, which has the same value as

```
(or (foo x) (zerop x))
```

where `foo` has the possibility of returning values other than `t` and `nil`.

2.4.4 The following two **and** expressions

(**and** ⟪condition$_1$⟫ ⟪condition$_2$⟫ ⟪condition$_3$⟫)

(**and** ⟪condition$_1$⟫ ⟪condition$_2$⟫ ⟪condition$_3$⟫ **t**)

do not always return the same value. When do they return different values?

2-5 Conditional Expressions

2.5.1 The **when** expression

(**when** ⟪condition⟫ ⟪expression⟫)

can be rewritten using an **and** expression. Show such an expression. Also, the **when** expression

(**when** (**and** ⟪condition$_1$⟫ ... ⟪condition$_n$⟫) ⟪expression⟫)

can be rewritten using one **and** expression. Show such expression.

2.5.2 Define a function **max3** which takes three numbers and returns the value of the biggest number of such three. Show the definition using **if** and the definition using **cond**.

2-6 Multiple-Valued Functions

2.6.1 When an expression returns or might return a multiple number of values, you can type

(**values** ⟪expression⟫)

in order to pick up only the first value. Using this method, define a function **floor-1** which returns only the first value of the **floor** function as its value.

2.6.2 Define a function **floor-2** which returns the second value of the **floor** function as its value.

2.6.3 Show the expression, using `multiple-value-list`, which has the same value as

```
(multiple-value-setq (x y) (foo))
```

What happens if you do not know how many values the function `foo` will return?

2.6.4 What values will be returned by the following expressions:

a. `(let ((x (values))) x)`
b. `(let ((x (values 1))) x)`
c. `(let ((x (values 1 2 3))) x)`
d. `(list (values) (values 1) (values 1 2 3))`

2.6.5 The function `quadratic-equation`, which can be used for solving quadratic equations, can be defined as follows (see Section 2.6 of *Introduction to Common Lisp*):

```
(defun quadratic-equation (b c)
  (let ((root (sqrt (- (* b b) (* 4 c)))))
    (if (zerop root)
        (/ (- b) 2)
        (values (/ (+ (- b) root 2)
                   (/ (- (- b) root 2)))))
```

Define the function `check-answers` which checks that `quadratic-equation` returns the correct values. The function `check-answers` should take two arguments, give them to `quadratic-equation`, assign the answer to the left side of the original equation, and return the result as a value. If a quadratic equation has two solutions, `check-answers` should return two values.

Answers to Exercises

2.1.1

```
(defun my-cadr (x) (car (cdr x)))
(defun my-cdddr (x) (cdr (cdr (cdr x))))
(defun my-cadadr (x) (car (cdr (car (cdr x)))))
```

2.1.2

```
(defun poly-2-3-4 (x)
  (+ (* 2 x x)
     (* 3 x)
     4))
```

It usually takes a longer time to do multiplication than addition. The above definition has three multiplications. In the following definition, there are only two multiplications.

```
(defun poly-2-3-4 (x)
  (+ (* (+ (* 2 x)
           3)
        x)
     4))
```

2.1.3

```
(defun area (r) (* pi r r))
```

or

```
(defun area (r) (* pi (expt r 2)))
```

2.2.1 It is a function which returns the positive difference between `x` and `y`, i.e., $|x - y|$.

2.2.2 The value of the expression is `(3 18 3 36 3)`. The following example should help you understand this:

```
(let ((v 3))                    ; v = 3
  (list v                       ; v = 3
        (let ((v (* v v)))      ; Since v = 3 × 3 = 9,
          (+ v v))              ;  v + v = 9 + 9 = 18
        v                       ; v = 3
```

```
(let ((v (+ v v)))  ; Since v = 3 + 3 = 6,
  (* v v))          ;   v × v = 6 × 6 = 36
v))                 ; v = 3
```

The above example is shown as an exercise. We do not recommend that you write such a program. It is hard to read and is likely to contain lots of errors because programs like this have local variables with the same name whose scopes overlap. Since you can use as many variable names as you want, you should use new variable names whenever possible. The following example should be much easier to read:

```
(let ((v 3))                ; v = 3
  (list v                   ; v = 3
        (let ((w (* v v)))  ; Since w = 3 × 3 = 9,
          (+ w w))          ;   w + w = 9 + 9 = 18
        v                   ; v = 3
        (let ((y (+ v v)))  ; Since y = 3 + 3 = 6,
          (* y y))          ;   y × y = 6 × 6 = 36
        v))                 ; v = 3
```

2.3.1

a. (T T NIL NIL NIL)
b. (NIL T T T NIL)
c. (NIL NIL NIL T T)

2.3.2

a. (NIL NIL T T NIL)
b. (NIL NIL T NIL T)
c. (NIL T NIL T NIL)
d. (T NIL NIL NIL T)
e. (T NIL NIL T NIL)

2.3.3

```
(defun minusp (x) (< x 0))
(defun zerop (x) (= x 0))
```

2.3.4

```
(defun my-evenp (x) (zerop (mod x 2)))
```

2.3.5

```
(defun adenoy (x)
  (if (evenp x) (/ x 2) (- x 1)))
```

2.4.1

```
(defun my-oddp (x) (not (zerop (mod x 2))))
```

2.4.2

```
(if (numberp x)
    (if (>= x 0)
        (<= x 100)
        nil)
    nil)
```

After you omit all `nil`s, you will have

```
(if (numberp x)
    (if (>= x 0) (<= x 100)))
```

2.4.3

```
(let ((temp (foo x)))
  (if temp temp (zerop x)))
```

2.4.4 When ⟪condition$_1$⟫ and ⟪condition$_2$⟫ are true, the first expression returns the value of ⟪condition$_3$⟫ while the second expression always returns `t` or `nil`.

2.5.1 The first `when` expression has the same value as

`(and` ⟪condition⟫ ⟪expression⟫ `)`

The next `when` expression has the same value as

`(and` ⟪condition$_1$⟫ ... ⟪condition$_n$⟫ ⟪expression⟫ `)`

2.5.2 If the `if` is used, it can be defined as

```
(defun max3 (x y z)
  (if (>= x y)
      (if (>= x z) x z)
      (if (>= y z) y z)))
```

If the `cond` is used, it is defined as

```
(defun max3 (x y z)
  (cond ((>= x y)
         (cond ((>= x z) z)
               (t z)))
        (t
         (cond ((>= y z) y)
               (t z)))))
```

2.6.1

```
(defun floor-1 (x y) (values (floor x y)))
```

2.6.2

```
(defun floor-2 (x y)
  (multiple-value-bind (q r) (floor x y)
    r))
```

2.6.3

```
(let ((temp (multiple-value-list (foo))))
  (setq y (cadr temp))
  (setq x (car temp)))
```

The important thing about this example is that even when `foo` returns two or fewer values, this expression should have the same values as the expression in Exercise 2.6.3. Also, the assignment to `x` is done later to make the whole `let` form return the same value as the `multiple-value-setq` form.

2.6.4 a. `NIL` b. `1` c. `1` d. `(NIL 1 1)`

2.6.5

```
(defun check-answers (b c)
  (multiple-value-bind (ans1 ans2)
                       (quadratic-equation b c)
    (if (null ans2)
        (+ (* ans1 ans1) (* b ans1) c)
        (values
          (+ (* ans1 ans1) (* b ans1) c)
          (+ (* ans2 ans2) (* b ans2) c)))))
```

3 Control Structure

Function call is the most basic control structure in Lisp. A function can call itself recursively.

To control Lisp programs, you can also use nonlocal exits in addition to function call. The `block` form and the `return-from` form supply static nonlocal exits while the `catch` form and the `throw` form supply dynamic nonlocal exits.

The forms `loop`, `do`, and `dolist` are used for iteration. The `tagbody` form and the `go` form, which provide for a "goto" facility, are also used for controlling a program.

- summary of functions

  `(trace` ⟪function name⟫ ... ⟪function name⟫ `)`
  `(untrace` ⟪function name⟫ ... ⟪function name⟫ `)`

  `(block` ⟪block name⟫ ⟪expression⟫ ... ⟪expression⟫ `)`
  `(return-from` ⟪block name⟫ `[`⟨value⟩`])`
  `(return [`⟨value⟩`])`
  `(loop` ⟪expression⟫ ... ⟪expression⟫ `)`
  `(tagbody` ⟨tag or expression⟩ ... ⟪tag or expression⟫ `)`
  `(go` ⟪tag⟫ `)`
  `(prog ((` ⟪variable⟫ ⟨initial value⟩`)`
  `...`
  `(` ⟪variable⟫ ⟨initial value⟩`))`
  ⟪tag or expression⟫ ... ⟪tag or expression⟫ `)`

```
(prog* ((⟪variable⟫ ⟨initial value⟩)
        ...
        ( ⟪variable⟫ ⟨initial value⟩))
   ⟪tag or expression⟫ ... ⟪tag or expression⟫ )

(endp ⟨list⟩)

(do (( ⟪variable⟫ [⟨initial value⟩ [⟪step⟫]])
     ...
     ( ⟪variable⟫ [⟨initial value⟩ [⟪step⟫]]))
  ( ⟪ending condition⟫ ⟪expression⟫ ... ⟪expression⟫ )
 ⟪expression⟫ ... ⟪expression⟫ )

(do* (( ⟪variable⟫ [⟨initial value⟩ [⟨step⟩]])
      ...
      ( ⟪variable⟫ [⟨initial value⟩[⟪step⟫]]))
   (⟪ending condition⟫ ⟪expression⟫ ... ⟪expression⟫ )
  ⟪expression⟫ ... ⟪expression⟫ )

(dolist (⟪variable⟫ ⟨list⟩ [⟨result⟩])
   ⟪expression⟫ ... ⟪expression⟫ )

(dotimes ( ⟪variable⟫ ⟨number⟩ [⟨result⟩])
   ⟪expression⟫ ... ⟪expression⟫ )

(catch ⟨catch tag⟩ ⟪expression⟫ ... ⟪expression⟫ )
(throw ⟨catch tag⟩ ⟨value⟩)
(unwind-protect ⟪expression⟫ ⟪expression⟫ ... ⟪expression⟫ )
```

3-1 Recursive Call

3.1.1 The following displays a trace of the function `my-evenp` which checks whether a nonnegative integer is even or not and the function `my-oddp` which checks if a nonnegative integer is odd or not. Fill in the appropriate values in the spaces [ ].

```
>(defun my-evenp (x)
   (if (zerop x) t (my-oddp (1- x))))
MY-EVENP
```

```
>(defun my-oddp (x)
   (if (zerop x) nil (my-evenp (1- x))))
MY-ODDP

>(trace my-evenp my-oddp)
(MY-EVENP MY-ODDP)

>(my-evenp 4)
  1> (MY-EVENP 4)
    2> (MY-ODDP 3)
      3> ([  ][  ])
        4> (MY-ODDP[  ])
          5> ([  ] 0)
          <5 ([  ] T)
        <4 (MY-ODDP [  ])
      <3 (MY-EVENP T)
    <2 (MY-ODDP [  ])
  <1 (MY-EVENP T)
 T

>
```

3.1.2 Use recursion to define the function `my-length` which determines the length of a list, that is, the number of the elements in a list.

```
>(my-length '(a b c))
3

>(my-length nil)
0
```

[Hints]

1. The length of an empty list is 0.
2. The length of a list other than an empty list is 1 plus the number of the elements of the list with the first element removed.

3.1.3 Use recursion to define the function `count-numbers` which takes a list and returns the number of numbers that appear in the list.

```
>(count-numbers '(a 1 b 2 c))
2
```

```
>(count-numbers nil)
0
```

[Hints]

1. If the argument is the empty list, then it returns 0.
2. When the argument is not the empty list,
   2.1 if the first element of the list is a number, add 1 to the number of numbers in the rest of the list;
   2.2 if the first element of the list is not a number, then return the number of numbers in the rest of the list.

3.1.4 Use recursion to define the function `append2` which takes two lists and returns the list of all the elements of the first list followed by all the elements of the second list.

```
>(append2 '(a b c) '(1 2 3 4))
(A B C 1 2 3 4)

>(append2 nil '(1 2 3 4))
(1 2 3 4)
```

3.1.5 Define the function `my-reverse` which takes a list and returns a list whose elements are in reverse order.

```
>(my-reverse '(a b c d))
(D C B A)

>(my-reverse nil)
NIL
```

We recommend the use of an auxiliary function, `my-reverse-append`, to define `my-reverse`. The function `my-reverse-append` takes two lists and returns the second list with the elements of the first list appended in reverse order.

```
>(my-reverse-append '(a b c) '(1 2 3 4))
(C B A 1 2 3 4)

>(my-reverse-append nil '(1 2 3 4))
(1 2 3 4)

>(my-reverse-append '(a b c) nil)
(C B A)
```

`my-reverse` can be defined using `my-reverse-append` as follows:

```
(defun my-reverse (x)
  (my-reverse-append x nil))
```

Define `my-reverse-append` recursively.

3.1.6 The Fibonacci array

Let's consider a numerical array $\{a_i\}(i = 0, 1, 2, \ldots)$ called the Fibonacci array. This numerical array is defined as

$$a_0 = a_1 = 1$$

$$a_n = a_{n-1} + a_{n-2} \qquad (n \geq 2)$$

Define the function `fib` which gets the value of a_n when it is given the index n of the Fibonacci array.

3.1.7 Tail-recursive call

Suppose a function `foo` is defined using a function `bar`. When `bar` is called from `foo` and the value of `bar` itself becomes the value of `foo`, we say that `foo` calls `bar` *tail-recursively.* For example, in

```
(defun cannot-help-it (x)
  (aaa x)
  (if (bbb x)
      (ccc)
      (ddd (eee x))))
```

`ccc` and `ddd` are called tail-recursively while `aaa`, `bbb`, and `eee` are not called tail-recursively. When a function calls itself tail-recursively, we call it a *self-tail-recursive* call. In the above example, if we replace `ddd` by `cannot-help-it`, `cannot-help-it` will be called self-tail-recursively. (Some people use self-tail-recursive and tail-recursive interchangeably.) Some Lisp compilers recently developed improve their efficiency by replacing a self-tail-recursive call with an iteration rather than with a function call. Let's look at the function which determines whether there are numbers among the elements of a list:

```
(defun list-numberp (x)
  (and (not (null x))
       (or (numberp (car x))
           (list-numberp (cdr x)))))
```

The above function is defined self-tail-recursively. In other words, if `x` is not an empty list and the `(car x)` is not a number, it calculates `(list-numberp (cdr x))` and takes this value as its own. When the KCL compiler processes this definition, `list-numberp` will be executed as follows:

1. The system prepares a box for a local variable `x` and puts the argument in this box;
2. If the value of `x` is an empty list, it returns `nil` and ends the execution;
3. If `(car x)` is a number, it returns `t` and ends the execution;
4. Otherwise it executes `(setq x (cdr x))` and goes back to 2.

Some functions which are defined recursively can be changed to self-tail-recursive functions. Let's look at the function `list-sum` (see Section 3.3 of *Introduction to Common Lisp*). The definition

```
(defun list-sum (x)
  (if (null x)
      0
      (+ (list-sum (cdr x)) (car x))))
```

is not self-tail-recursive because it adds the value of `(list-sum (cdr x))` to `(car x)` rather than returning the value of `(list-sum (cdr x))` itself. We can redefine this function to be self-tail-recursive as follows:

```
(defun revised-list-sum (x)
  (list-sum-aux x 0))

(defun list-sum-aux (x so-far)
  (if (null x)
      so-far
      (list-sum-aux (cdr x)
                    (+ (car x) so-far))))
```

The function `revised-list-sum` starts adding the list from the beginning while `list-sum` in the first definition executes addition of the list from the end. The auxiliary function `list-sum-aux` takes, as a parameter, `so-far`, which is the partial sum of the elements added up so far, and computes the next partial sum by adding `(car x)` to the sum. When the list is empty, `so-far` is the total value. When `list-sum-aux` is first called from `revised-list-sum`, the partial sum is 0. The recursive call of `list-sum-aux` is self-tail-recursive.

In order to see the difference in execution speed between `list-sum` and `list-sum-aux`, we ran the following test program. We built a list of 1000 elements of all 1's.

```
>(compile 'list-sum)          ; compile list-sum
LIST-SUM

>(time (list-sum x))          ; check the execution time
real time : 0.000 secs        ; of (list-sum x)
run time  : 0.042 secs        ; it took 0.042 seconds
1000

>(compile 'list-sum-aux)      ; compile list-sum-aux
LIST-SUM-AUX

>(time (list-sum-aux x 0))    ; check the execution time
real time  : 0.000 secs       ; of list-sum aux x 0
run time   : 0.028 secs       ; it took 0.028 seconds
1000

>
```

In the above example, the execution time of `list-sum-aux` is only half the execution time of `list-sum`. In actual programs, the difference may not be as extreme as it is in this example, but the function call time will certainly be shortened by using a self-tail-recursive call.

The function `fact` for factorial can be defined as follows (see Section 3.1 of *Introduction to Common Lisp*):

```
(defun fact (n)
  (if (zerop n)
      1
      (* n (fact (1- n)))))
```

Redefine the function `fact` to be self-tail-recursive as in the above example.

3.1.8 Tower of Hanoi (1)

Let us solve the puzzle called the **Tower of Hanoi**. There are three poles and some disks on Pole A. (See Figure 3.1) The problem is to move all the disks to Pole C by moving one disk at a time without putting any disk on top of a smaller disk.

When there are three disks, the problem can be solved by moving as in Figure 3.2. (Disks are numbered with the smallest getting 1.) Let us define a function `Hanoi` which shows how to move n disks for any given integer `n`.

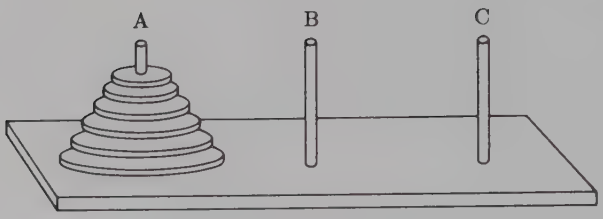


Figure 3.1 Tower of Hanoi

```
>(Hanoi 3)
Move disk 1 from A to C.
Move disk 2 from A to B.
Move disk 1 from C to B.
Move disk 3 from A to C.
Move disk 1 from B to A.
Move disk 2 from B to C.
Move disk 1 from A to C.
NIL
```

We will use the following function for the screen display:

```
(defun move-disk (disk from to)
  (format t "~&Move disk ~A from ~A to ~A."
          disk from to))
```

Here, the parameter `disk` is the number written on the disk and the parameters `from` and `to` are one of the three symbols, `a`, `b`, or `c`. (Please refer to Section 11.9 of *Introduction to Common Lisp* for an explanation of the `format` form).

```
>(move-disk 1 'a 'b)
Move disk 1 from A to B.
NIL
```

Since it may be too difficult for some readers to solve this problem, we will give you some hints. But if you think you can solve this without hints, please try to do so.

When $n = 1$, you only have to move disk 1 from A to C. When $n = 2$, you should move disk 1 to B first, move disk 2 to C, and then move disk 1 from B to C. How about when $n = 3$? In order to move disk 3, which is the biggest disk, to C, you need to move two other disks to B first. After moving disk 3 to C, you only need to move two disks from B to C. You

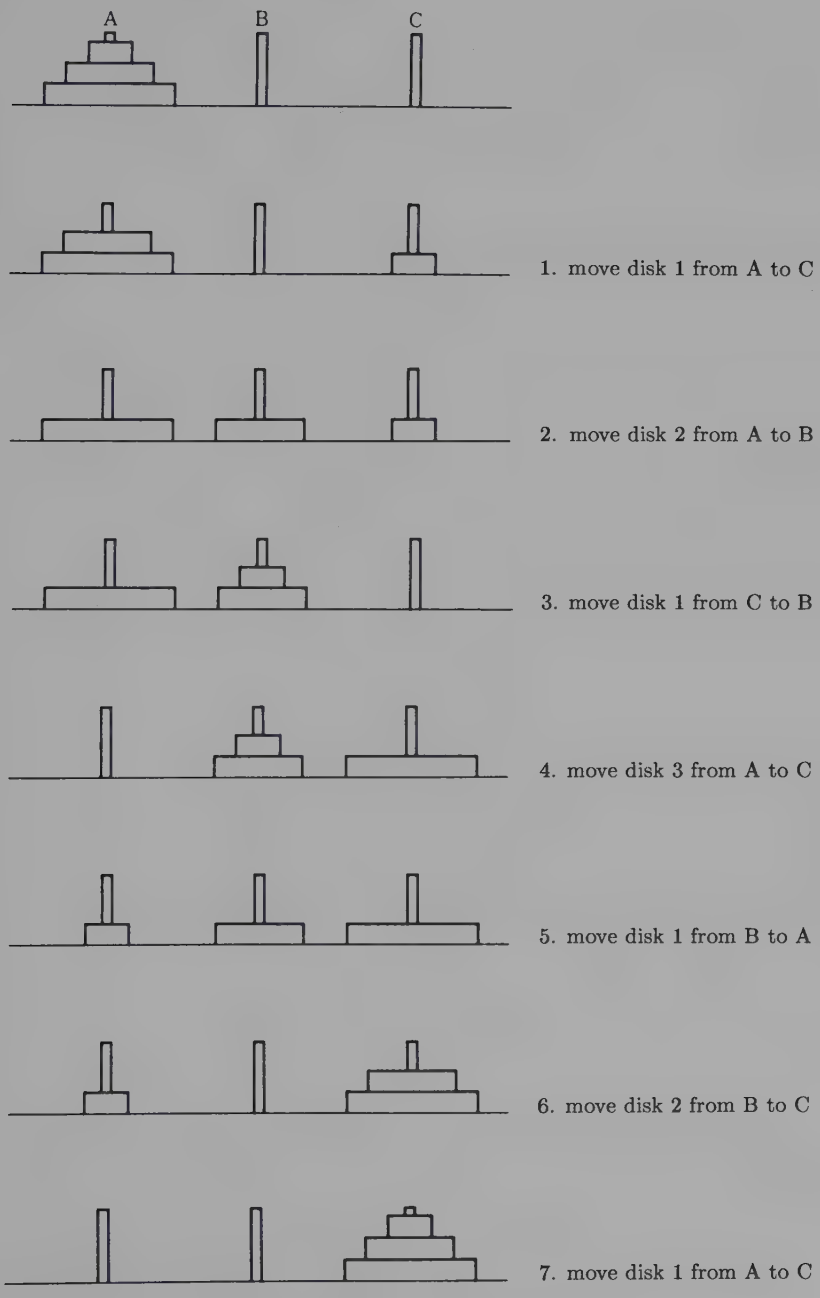


Figure 3.2 Solution to Tower of Hanoi

can do this by using a method similar to solving `(Hanoi 2)`. The only difference here is that you need to move disks from A to B and from B to C. When $n \geq 4$, the idea is the same. You need to move the $n-1$ disks above disk n to B, move disk n to C, and then move other disks from B to C using the method used in solving `(Hanoi (1- n))`.

3.1.9 Tower of Hanoi (2)
The name "Tower of Hanoi" comes from an ancient story. Long time ago, some monks were trying to solve this puzzle using 64 disks. According to them, the end of the world would come when the puzzle is solved. Figure out when this will be. First, define the function `time-Hanoi` which calculates the number of times a disk needs to be moved in solving `(Hanoi n)`. Then, supposing it takes one second to move one disk, determine when this puzzle will be solved. One year has 365 days and the monks will work continuously without stopping. It is not clear when the monks started to try to solve the puzzle; however, these monks could not have existed before the birth of Buddha. So, let's suppose they started 500 B.C.

3-2 Block Structure

3.2.1 The function `list-sum` (see Exercise 1.3.7) can be defined using the `loop` form as follows:

```
(defun list-sum (x)
  (let ((sum 0))
    (loop (when (null x) (return sum))
          (setq sum (+ (car x) sum))
          (setq x (cdr x)))))
```

You will see the following patterns often when using repetitive control such as `loop`:

```
(setq x (+ x ⟨amount of increase⟩))
(setq x (- x ⟨amount of decrease⟩))
(setq (cdr x))
(setq x (cond ⟨value⟩ x))
```

You can use such patterns, but the program will look cleaner if you use the following macros. Since you do not have to specify the same variable more than once, it also prevents mistyping.

```
(incf x [⟨amount of increase⟩])
(decf x [⟨amount of decrease⟩])
(pop x)
(push ⟨value⟩ x)
```

I recommend the use of such macros.

```
(incf x)
(decf x)
```

has the same value as

```
(incf x 1)
(decf x 1)
```

`list-sum` is defined using such functions as follows:

```
(define list-sum (x)
  (let ((sum 0))
    (loop (when (null x) (return sum))
          (incf sum (car x))
          (pop x))))
```

Define the function `my-length` (See Exercise 3.1.2) using a `loop` expression. Use the above-mentioned macros as much as possible.

3.2.2 Define the function `list-numberp` (see Exercise 3.1.7) using a `loop` expression.

3.2.3 Define the function `my-reverse` (see Exercise 3.1.5) using a `loop` expression.

3.2.4 The read-eval-print loop (1)
Let me remind you that the top level of a Lisp system repeats the following operations:

1. displays a prompt;
2. reads one form;
3. evaluates the form; and
4. displays its result.

This iteration is called a **read-eval-print loop**. Let us define a function which imitates this loop.

```
(defun read-eval-print-loop ()
  (loop (print 'eval>)
        (print (eval (read)))))
```

In order to distinguish this prompt from the prompt of the read-eval-print loop used by the system, we use `EVAL>` as a prompt for this function. `read` is a system function which reads one Lisp datum. The above function gives `eval` the datum which is read by `read` and prints the result. In this way it does steps 1 through 4 mentioned above. This definition is very simple but it works.

```
>(read-eval-print-loop)

EVAL>(- (* 2 4) 3)

5
EVAL>(setq x 3)

3
EVAL>(* x x)

9
EVAL>
```

The screen display can be improved but to do this you need to understand Chapter 11 of this book, so leave it this way at the moment.

Many Lisp systems actually define and use functions like `read-eval-print-loop`. For example, in KCL, the system automatically calls a function like `read-eval-print-loop` when it is started. (The name of this function is a secret.) However, the definition of such a function in an actual system is much longer than this example (about thirty lines in KCL).

The above mentioned `read-eval-print-loop` does not have a good way to get out of the loop; there are ways but they should only be used in an emergency. Change the definition so that when `nil` is input as top-level form, the system stops the execution of `read-eval-print-loop`.

```
>(read-eval-print-loop)
EVAL>(cons 'a 'b)

(A . B)
EVAL>nil
NIL

>
```

Redefine the definition of `read-eval-print-loop` to do this.

3-3 Program Structure

3.3.1 It is better not to use the program structure functions `go` and `tagbody` (see Section 3.3 in *Introduction to Common Lisp*) more than necessary. Using `go` when branching complicates the flow of the control and decreases the readability of a program. It also introduces additional possibilities for making program errors. Common Lisp has better structured control functions, and you should usually write a program using such control functions rather than using `go`.

Among the special forms of many Common Lisp systems, the combination of `tagbody` and `go` is the only primitive function which can be used to do iteration. This is the reason we described the program structure functions in *Introduction to Common Lisp.*

Actually, many Common Lisp systems expand the system macro expressions such as `loop` and `do` to an expression using `go`. We mention this so that a user will not be surprised at seeing `go` when they use `macroexpand` on these iterative forms.

With the above-mentioned information in mind, fill in `[ ]` in the following sentences:

a. `go` is the `[ ]` method to try to use.
b. The person who uses `go` lacks `[ ]`.
c. `go` has `[ ]` but no `[ ]`.

3-4 Macros for Iteration Constructs

3.4.1 `list-sum` was defined using `loop` in Exercise 3.2.1. This function can be defined using a `do` expression without a body as follows:

```
(defun list-sum (x)
  (do ((y x (cdr y))
       (sum 0 (+ sum (car y))))
      ((endp y) sum)))
```

The `do` in this expression cannot be replaced by `do*`. Explain why. Also explain what needs to be done in order to use `do*`.

3.4.2 Redefine `list-sum` (defined in Exercise 3.4.1) using `dolist` rather than `do`.

3.4.3 Show three definitions of `my-length` (defined in Exercise 3.2.1) using `do`, `do*`, and `dolist`, respectively.

3.4.4 Show three definitions of `list-numberp` (defined in Exercise 3.2.2) using `do`, `do*`, and `dolist`, respectively.

3.4.5 Show three definitions of `my-reverse` (defined in Exercise 3.2.3) using `do`, `do*`, and `dolist`, respectively.

3.4.6 Using `dotimes`, define the function `my-make-list` which takes a nonnegative integer `n` and some data `x` and returns a list of length `n` whose elements are all `x`.

```
>(my-make-list 5 '(a b c))
((A B C)(A B C)(A B C)(A B C)(A B C))

>(my-make-list 3 'good-bye)
(GOOD-BYE GOOD-BYE GOOD-BYE)

>(my-make-list 0 'none)
NIL
```

3.4.7 read-eval-print loop (2)
The function `read-eval-print-loop` defined in Exercise 3.2.4 cannot deal with a top level form that returns more than one value.

```
>(read-eval-print-loop)

EVAL>(values)

NIL
EVAL>(values 1)

1
EVAL>(values 1 2)

1
```

Redefine `read-eval-print-loop` so that it displays all the values which are returned by a top level form.

```
>(read-eval-print-loop)

EVAL>(floor 9 4)
```

```
2
1
EVAL>(values)

EVAL>(+ 2 3)

5
EVAL>nil
NIL

>
```

3-5 catch and throw

3.5.1 Functions foo and bar are defined as follows:

```
(defun foo (x)
  (let ((v (catch 'top (bar x 1))))
    (case (car v)
      (not-a-number
        (cons 'the
              (cons (cadr v)
                    '(th element is not a number))))
      (negative
        (cons 'the
              (cons (cadr v)
                    '(th element is negative))))
      (otherwise '(every element is
                   nonnegative number)))))

(defun bar (x n)
  (cond ((endp x) (throw 'top nil))
        ((not (numberp (car x)))
         (throw 'top (list 'not-a-number n)))
        ((minusp (car x))
         (throw 'top (list 'negative n)))
        (t (bar (cdr x) (1+ n)))))
```

Using the above definitions, find the values of the following expression:

a. `(foo '(1 2 3 4 5))`
b. `(foo '(1 2 3 four 5))`
c. `(foo '(1 2 3 -4 5))`

3.5.2 read-eval-print loop (3)

In Exercise 3.2.4, we input `nil` to end the `read-eval-print-loop`, but that is not what Lisp is supposed to do. When `nil` is input, it should display `nil` and continue the interaction. Let's make a function called `good-bye` for ending the interaction. You should be able to call `good-bye` anytime during the execution of the `read-eval-print-loop`. When `good-bye` is called, the system should immediately end the execution of the `read-eval-print-loop` and return the first argument of `good-bye` as a value.

```
>(read-eval-print-loop)

EVAL>(list 1 2 3)

(1 2 3)
EVAL>(good-bye 100)
100

>(read-eval-print-loop)

EVAL>(defun foo (x)
       (when (numberp x)
         (good-bye x)))

FOO
EVAL>(foo 'bar)

NIL
EVAL>(foo 3)
3

>
```

Define the function `good-bye` and define `read-eval-print-loop` again to make use of the function `good-bye`.

3.5.3 read-eval-print loop (4)

Improve `read-eval-print-loop` in Exercise 3.5.2 so that the system displays a message every time the execution ends.

```
>(read-eval-print-loop)

EVAL>(good-bye 3)

GOOD-BYE
3

>(read-eval-print-loop)

EVAL>atcholi
Error: The variable ATCHOLI is unbound.
Error signalled by EVAL.

Broken at EVAL.  Type :H for Help.
>>:q

GOOD-BYE

Top level.
>(catch 'a-catcher
   (read-eval-print-loop))

EVAL>(throw 'a-catcher '(good bye))

GOOD-BYE
(GOOD BYE)

>
```

Answers to Exercises

3.1.1

```
>(my-evenp 4)
  1> (MY-EVENP 4)
    2> (MY-ODDP 3)
      3> ([MY-EVENP] [2])
        4> (MY-ODDP [1])
          5> ([MY-EVENP] 0)
          <5 ([MY-EVENP] T)
        <4 (MY-ODDP [T])
      <3 (MY-EVENP T)
    <2 (MY-ODDP [T])
  <1(MY-EVENT T)
T
```

3.1.2

```
(defun my-length (x)
  (if (null x)
      0
      (1+ (my-length (cdr x)))))
```

3.1.3

```
(defun count-numbers (x)
  (if (null x)
      0
      (if (numberp (car x))
          (1+ (count-numbers (cdr x)))
          (count-numbers (cdr x)))))
```

3.1.4

```
(defun append2 (x y)
  (if (null x)
      y
      (cons (car x) (append2 (cdr x) y))))
```

3.1.5

```
(defun my-reverse-append (x y)
  (if (null x)
      y
      (my-reverse-append (cdr x) (cons (car x) y))))
```

3.1.6

```
(defun fib(n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

The above definition is not efficient because it calculates the same value many times. For example, when the system processes `(fib 5)`, it processes `(fib 4)` and `(fib 3)`. It also processes `(fib 3)` during the calculation of `(fib 4)`. Therefore, `(fib 3)` is processed twice. As the value of `n` gets bigger, the number of repetitions increases. When the system processes `(fib 6)`, it processes `(fib 3)` four times. When the system processes `(fib 7)`, it processes `(fib 3)` eight times. For `(fib 8)`, the system processes it sixteen times. For `(fib 1000)`, it processes `(fib 3)`

```
>(expt 2 996)
669692879491417075592765655662501131600878007 3
159585046523439927314694069530850765582489867 5
980991132974667057347071676574196580355769627 7
249036098418660925245910485926514436588817162 8
163981963673721363845654046864738713292124229 7
244784649662981643216069977985540888547877686 4
```

times. You can eliminate this repetition by defining

```
(defun fib (n) (values (fib-aux n)))
```

and by using an auxiliary function which returns two values

```
(defun fib-aux (n)
  (if (< n 2)
      (values 1 1)
      (multiple-value-bind (prev prev-prev)
                           (fib-aux (1- n))
        (values (+ prev prev-prev) prev))))
```

3.1.7 fixnum and bignum

```
(defun fact (n) (fact-aux n 1))

(defun fact-aux (n so-far)
  (if (zerop n)
      so-far
      (fact-aux (1- n) (* n so-far))))
```

We have compiled the above definition and the definition in *Introduction to Common Lisp* using KCL and compared the execution times. In order to avoid confusion, I called the definition in *Introduction to Common Lisp* `fact-original`.

```
(defun fact-original (n) .
  (if (zerop n)
      1
      (* n (fact-original (1- n)))))
```

The result is the following:

```
>(time (fact 100))
real time : 0.000 secs
run time  : 0.067 secs
...

>(time (fact-original 100))
real time : 0.000 secs
run time  : 0.057 secs
...
```

It turns out that the self-tail-recursive call took a longer time. This is not because the compiler has some defect. It is because the original definition and the self-tail-recursive definition used a different method of multiplying.

In `fact-original`, in order to compute the nth multiplication, the system multiplies

$$1 \times 2 \times 3 \times \cdots \times n$$

In the self-tail-recursive function the system counts down from n.

$$n \times (n-1) \times \cdots \times 2 \times 1$$

The order of numbers multiplied by the system is reversed, and this difference in the order has affected the time of execution. In order to confirm this, we ran the following experiment:

```
>(time (* 100 99 98 ... 4 3 2 1))
real time : 0.000 secs
run time  : 0.067 secs
...

>(time (* 1 2 3 4 ... 98 99 100))
real time : 0.000 secs
run time  : 0.057 secs
...
```

By looking at these results, it is clear that, in both `fact` and `fact-original`, the system spent hardly any time doing the recursive call or iteration but spent almost all of the execution time for multiplication.

A Common Lisp system can treat any integers no matter how big they are. However, the system does not do calculations on big integers with the same efficiency as the calculations involving small integers. The system multiplies or adds integers whose absolute values are relatively small faster than bigger numbers by using the capabilities of the hardware. Such relatively small integers are called **fixnum**s, and integers other than fixnums are called **bignum**s. The range of fixnums varies on different computer systems. Integers which are smaller than the constant variable called the `most-positive-fixnum` and which are bigger than the constant variable called the `most-negative-fixnum` are the fixnums of the system. In KCL

```
>most-negative-fixnum
-2147483648

>most-positive-fixnum
2147483647
```

Let's go back to `fact` and `fact-original`. In `fact-original`, since the multiplication starts with 1, the multiplication of fixnums lasts for a while. In `fact`, the multiplication starts from the biggest number, and the value of the multiplication reaches a bignum quite soon. From there on each multiplication will be a bignum multiply. The multiplications in `fact-original` take a shorter time because of the smaller number of bignum multiplications.

3.1.8

```
(defun Hanoi (n)
  (Hanoi-aux n 'a 'c 'b))

(defun Hanoi-aux (n from to free)
  (if (= n 1)
      (move-disk 1 from to)
      (progn
        (Hanoi-aux (1- n) from free to)
        (move-disk n from to)
        (Hanoi-aux (1- n) free to from))))
```

The parameter called `from` in the auxiliary function `Hanoi-aux` is the post on which the n disks start, `to` is the post to which the disks are to be moved, and `free` is the post in the middle. The three expressions in the `progn` do the following:

1. move the disks other than the disk n;
2. move the disk n to the appropriate post;
3. move the rest of the disks to the appropriate post.

If you imagine that when n is 0, the expressions should not do anything, the definition of `Hanoi-aux` will look like

```
(defun Hanoi-aux (n from to free)
  (when (> n 0)
    (Hanoi-aux (1- n) from free to)
    (move-disk n from to)
    (Hanoi-aux (1- n) free to from)))
```

3.1.9 The answer to Exercise 3.1.8 can be rewritten in simpler definition:

```
(defun time-Hanoi (n)
  (time-Hanoi-aux n 'a 'c 'b))
```

```
(defun time-Hanoi-aux (n from to free)
  (if (= n 1)
      1
      (+ (time-Hanoi-aux (1- n) from free to)
         1
         (time-Hanoi-aux (1- n) free to from))))
```

This is correct, but since the values of

```
(time-Hanoi-aux (1- n) from free to)
```

and

```
(time-Hanoi-aux (1- n) free to from)
```

are the same, there are duplicate calculations (see the answer to Exercise 3.1.6). Since `time-Hanoi` has nothing to do with moving disks from where to where, it can be defined without using an auxiliary function as

```
(defun time-Hanoi (n)
  (if (= n 1)
      1
      (1+ (* (time-Hanoi (1- n)) 2))))
```

It takes

```
>(time-Hanoi 64)
18446744073709551615
```

seconds to solve the puzzle. Since one year has

```
>(* 365 24 60 60)
31536000
```

seconds, it will take

```
>(values (floor (time-Hanoi 64) (* 365 24 60 60)))
584942417355
```

years to solve the puzzle. Suppose the monks started this puzzle in 500 B.C., they will not finish the puzzle until the year

```
>(+ (floor (time-Hanoi 64) (* 365 24 60 60))
    -500)
584942416855
```

According to this calculation, the end of the world will come in about 500,000,000,000 (Five hundred thousand million) A.D. We do not have to worry.

3.2.1

```
(defun my-length (x)
  (let ((n 0))
    (loop (when (null x) (return n))
          (incf n)
          (pop x))))
```

3.2.2

```
(defun list-numberp (x)
  (loop (cond ((null x) (return nil))
              ((numberp (car x)) (return t))
              (t (pop x)))))
```

3.2.3

```
(defun my-reverse (x)
  (let ((result nil))
    (loop (when (null x) (return result))
          (push (car x) result)
          (pop x))))
```

3.2.4

```
(defun read-eval-print-loop ()
  (loop (print 'eval>)
        (let ((form (read)))
          (if (null form)
              (return)
              (print (eval form))))))
```

3.3.1

a. go is the [last] method to try to use. ([bad] or [poor] would be ok.)
b. The person who uses go lacks [ability in programming]. ([intelligence] would be ok.)
c. go has [disadvantages] but no [advantages].

3.4.1 Using `do*`, the initial value of `y` and `sum` would be the same as in the definition using `do`. However, before the next iteration, the value of `y` changes to `(cdr x)`, and then the value of `(car y)` is added to `sum`. As a result, the value of `(car x)` will not be added to `sum`. Also, at the end of the iteration, the value of `y` would become `nil` and then the system would try to process `(+ sum (car y))`. Unfortunately, the fact that `(car y)` is `nil` and not a number causes an error. In other words, while `(list-sum '(1 2 3 4))` adds the elements of `(1 2 3 4)` in the correct definition using `do`, it adds the elements of `(2 3 4 nil)` in the definition using `do*`.

In order to use `do*`, you just have to change the order of specifying the local variables.

```
(defun list-sum (x)
  (do* ((sum 0 (+ sum (car y)))
        (y x (cdr y)))
       ((endp y) sum)))
```

The above definition will produce the same result even if you replace `do*` with `do`.

3.4.2

```
(defun list-sum (x)
  (let ((sum 0))
    (dolist (n x sum)
      (incf sum n))))
```

In the above definition, you should not put the `let` expression inside of the main body of `dolist` because `sum` would be rebound each time the main body of the `dolist` is executed.

3.4.3 This function can be defined using `do` as follows:

```
(defun my-length (x)
  (do ((y x (cdr y))
       (n 0 (1+ n)))
      ((endp y) n)))
```

In the above definition, the result will be the same whether you use `do` or `do*`. It can be redefined using `dolist` as

```
(defun my-length (x)
  (let ((n 0))
    (dolist (e x n)
      (incf n))))
```

Since the execution of `my-length` does not depend on what the elements of `x` are, the local variable `e` is never used.

3.4.4 `list-numberp` can be defined using `dolist` as

```
(defun list-numberp (x)
  (dolist (e x nil)
    (when (numberp e) (return t))))
```

Since `dolist` builds the `nil` block, the purpose of the `(return t)` is to finish the iteration of the `dolist` and to return `t` as the value of the `dolist`. If there is no nonlocal jumping caused by the `return` form, and the `dolist` ends its iteration because no numerical element is found, the value of the `dolist` (thus the value of `list-numberp`) is `nil`. This is specified by the `nil` in `(e x nil)` right after the symbol `dolist`. This `nil` can be omitted, but it is better not to omit it when the value of the whole `dolist` expression has some meaning, such as in the above example.

It is better to use `dolist` than `do` to solve this example. Generally speaking, if `dolist` can be used, the program looks better if you use `dolist` rather than `do` or `do*`.

For your reference, here is the definition using `do`:

```
(defun list-numberp (x)
  (do ((y x (cdr y)))
      ((endp y) nil)
    (when (numberp (car y)) (return t))))
```

You may use `do*` instead of `do`.

3.4.5 Using `dolist` you can define `my-reverse` as follows:

```
(defun my-reverse (x)
  (let ((result nil))
    (dolist (e x result)
      (push e result))))
```

or you can define it as follows using do:

```
(defun my-reverse (x)
  (do ((result nil (cons (car y) result))
       (y x (cdr y)))
      ((endp y) result)))
```

Once again, you may use do* instead of do.

3.4.6

```
(defun my-make-list (n x)
  (let ((result nil))
    (dotimes (i n result)
      (push x result))))
```

3.4.7

```
(defun read-eval-print-loop ()
  (loop
    (print 'eval>)
    (let ((form (read)))
      (if (null form)
          (return)
          (dolist (value (multiple-value-list
                          (eval form)))
            (print value))))))
```

3.5.1

a. (EVERY ELEMENT IS NONNEGATIVE NUMBER)
b. (THE 4 TH ELEMENT IS NOT A NUMBER)
c. (THE 4 TH ELEMENT IS NEGATIVE)

3.5.2

```
(defun read-eval-print-loop ()
  (catch 'good-bye-bye-bye
    (loop ... )))

(defun good-bye (x)
  (throw 'good-bye-bye-bye x))
```

where (loop ...) is exactly the same as in the answer to Exercise 3.4.7.

3.5.3

```
(defun read-eval-print-loop ()
  (unwind-protect
    (catch 'good-bye-bye-bye
      (loop
        (print 'eval>)
        (let ((form (read)))
          (if (null form)
              (return)
              (dolist (value
                       (multiple-value-list
                         (eval form)))
                (print value))))))
    (print 'good-bye)))
```

4 Symbols and Packages

All symbols have print names. Print names are usually made up of capital letters, but they can also include lowercase letters or special letters. A print name is associated with the symbol by using the idea of a package. Objects of the type symbol which belong to two different packages are considered to be different symbols even if they have the same print name.

All of the function and variable names that are defined in Common Lisp are external symbols of the `lisp` package. Symbols made by a user usually become internal symbols of the `user` package, but they can be registered in another package by changing the current package or by specifying the package to a function which creates symbols. A symbol can become an external symbol by `export`ing it. You can also `import` an individual symbol of another package or `use` all the external symbols as a whole, or you can `shadow` the symbol which has been `import`ed. A symbol usually belongs to one home package, but you can create a symbol which does not belong to any package.

Each symbol of the system package, `keyword`, when used as a variable, is a constant which has itself as a value.

- summary of functions

  `(symbol-name` ⟨symbol⟩`)`
  `(symbol-package` ⟨symbol⟩`)`
  `(find-package` ⟨package name⟩`)`
  `(in-package` ⟨package name⟩`)`
  `(use-package` ⟨package name⟩ `[`⟨package⟩`])`
  `(export` ⟨symbol or symbol list⟩ `[`⟨package⟩`])`

```
(import ⟨symbol or symbol list⟩ [⟨package⟩])
(shadow ⟨symbol⟩ [⟨package⟩])

(intern ⟨string⟩ [⟨package⟩])
(find-symbol ⟨string⟩ [⟨package⟩])
(gensym [⟨number or string⟩])
(gentemp [⟨string⟩ [⟨package⟩]])

(do-symbols ( ⟪variable⟫ [⟨package⟩ [⟨result⟩]])
    ⟪expression⟫ ... ⟪expression⟫ )
(do-external-symbols ( ⟪variable⟫ [⟨package⟩ [⟨result⟩]])
    ⟪expression⟫ ... ⟪expression⟫ )
(do-all-symbols ( ⟪variable⟫ [⟨result⟩])
    ⟪expression⟫ ... ⟪expression⟫ )
(list-all-packages)
```

- summary of variables

```
*print-case*
*package*
```

4-1 Symbol Names and Print Names

4.1.1 What are the values of the following expressions:

a. `(symbol-name 'cons)`
b. `(symbol-name 'ConS)`
c. `(symbol-name '|cons|)`
d. `(symbol-name '|CONS|)`
e. `(symbol-name 'c\Ons)`
f. `(symbol-name 'c\ons)`

4.1.2 What are the values of the following expressions:

a. `(eq 'cons 'ConS)`
b. `(eq 'cons '|cons|)`
c. `(eq 'cons '|CONS|)`
d. `(eq 'cons 'c\Ons)`
e. `(eq 'cons 'c\ons)`

4.1.3 In *Introduction to Common Lisp*, symbols are printed using lowercase. In this book uppercase letters are used. Explain why.

4-2 Packages

4.2.1 The system function `list-all-packages` returns a list of all the existing packages and takes no arguments. Check what packages are part of your Common Lisp system.

4-3 Internal Symbols and External Symbols

4.3.1 The system predicate `fboundp` takes a symbol and returns `t` if the symbol is the name of either a special form, a macro, or a function and returns `nil` otherwise.

```
>(fboundp 'if)
T

>(fboundp 'do-external-symbols)
T

>(fboundp 'car)
T

>(fboundp 'new-symbol)
NIL

>(fboundp '*package*)
NIL

>(fboundp 'square)
NIL

>(defun square (x) (* x x))
SQUARE

>(fboundp 'square)
T
```

As described in *Introduction to Common Lisp*, the special form names, macro names, and function names which are defined by Common Lisp are all external symbols of the `lisp` package. Make a list of such symbols using `fboundp`.

4.3.2 The system predicate called `boundp` checks whether a symbol has a value as a global variable or not (see Section 9.1 in *Introduction to Common Lisp*).

```
>(boundp '*package*)
T

>(boundp 'rudolph)
NIL

>(setq rudolph 10)
10

>(boundp 'rudolph)
T
```

All the variable names which Common Lisp defines are registered as external symbols of the `lisp` package, and they have some initial values. Make a list of such variable names.

4.3.3 The system predicate `constantp` takes a symbol and determines whether the symbol is a constant variable or not. For example,

```
>(constantp 'pi)
T

>(constantp 't)
T

>(constantp '*print-case*)
NIL
```

Make an expression whose value is a list of the names of the constant variables defined by Common Lisp.

4.3.4 The following interaction shows how to change the current package to a new package called `fresh`. What is the value of the `let` expression?

```
>(in-package 'fresh)
#<"FRESH" package>

FRESH>(let ((x nil))
        (do-symbols (s) (push s x))
        x)
```

4-4 Interning

4.4.1 We would like to distinguish two Johns, John McCarthy at Stanford University and Pope John at the Vatican. Fill in the appropriate values in the spaces `[ ]` in the following interaction:

```
>(in-package 'stanford)
#<"STANFORD" package>

STANFORD>(setq john 42)
42

STANFORD>(in-package 'vatican)
#<"VATICAN" package>

VATICAN>(setq john 23)
23

VATICAN>john
[ ]

VATICAN>stanford::john
[ ]

VATICAN>(in-package 'stanford)
#<"STANFORD" package>

STANFORD>john
[ ]
```

```
STANFORD>vatican::john
[ ]

STANFORD>(in-package 'user)
#<"USER" package>

>stanford::john
[ ]

>vatican::john
[ ]

>(symbol-name 'stanford::john)
[ ]

>(symbol-name 'vatican::john)
[ ]
```

4.4.2 Define a function `my-symbol` which takes a symbol as an argument, makes a symbol whose print name has "`MY-`" at the front, and registers it in the current package.

```
>(my-symbol 'bonny)
MY-BONNY
```

In order to make a new string using some existing strings, you should use `concatenate` (see Section 10.2 in *Introduction to Common Lisp*) as in

`(concatenate 'string` $\langle \text{string}_1 \rangle \ldots \langle \text{string}_n \rangle$`)`

For example,

```
>(concatenate 'string "to" "get" "her")
"together"
```

4.4.3 Although Common Lisp does not have such a function, some Lisp languages have a function called `explode`. Its specification depends on the version of Lisp. Suppose we use the following specification: `explode` takes a symbol, makes new symbols one for each letter in the name of the symbol, registers them in the current package, and returns the list of these symbols as its value.

```
>(explode 'cons)
(C O N S)

>(explode '|Ab1|)
(A |b| |1|)
```

In order to define **explode**, it is necessary to have the following functions which can process strings.

The function **length**, which is used to determine the length of the string, can be used to determine the number of the letters in a string.

```
>(length "abcde")
5
```

The function **subseq** (see Section 10.2 in *Introduction to Common Lisp*) can be used to extract a part of a string.

(subseq ⟨string⟩ ⟨starting⟩ ⟨ending⟩)

subseq creates a string made of letters between the ⟨starting⟩ number and the ⟨ending⟩ number -1. We are assuming here that the first letter in ⟨string⟩ is 0th and the next letter is 1st, etc.

```
>(subseq "abcde" 2 4)
"cd"

>(subseq "abcde" 0 1)
"a"
```

Define **explode** using the above functions.

4.4.4 In Exercise 4.4.3, we defined a function **explode**. Define a function called **implode** which does the opposite operation to **explode**.

```
>(implode '(c o n s))
CONS

>(implode '(a \b \1))
|Ab1|
```

4-5 Predefined Packages

4.5.1 What are the values of the following expressions:

a. `(constantp ':test)`
b. `(constantp :test)`
c. `(symbol-value ':test)`
d. `(symbol-value :test)`
e. `(symbol-name :test)`

See Exercise 4.3.3 for information about `constantp`.

4-6 Shadowing

4.6.1 Fill in [ ] in the following interaction:

```
>(shadow 'car)
T

>(defun car (x) (cons 'car x))
CAR

>(car '(a b c))
[ ]

>(lisp:car '(a b c))
[ ]

>(shadow 'cdr)
T

>(defun cdr (x) (lisp:car x))
CDR

>(cdr '(a b c))
[ ]

>(lisp:cdr '(a b c))
[ ]
```

4-7 Functions for Making Symbols

4.7.1 Define a function **expand-or** which takes a list which starts with the symbol **or** and returns an expression which uses **let** and **if** and has the same value as the **or**.

```
>(expand-or '(or))
NIL

>(expand-or '(or x))
X

>(expand-or '(or x y))
(LET ((#:G834 X)) (IF #:G834 #:G834 Y))

>(expand-or '(or x y z))
(LET ((#:G835 X))
  (IF #:G835
      #:G835
      (LET ((#:G836 Y)) (IF #:G836 #:G836 Z))))
```

The symbols such as **#:G834** are created by **gensym**.

Answers to Exercises

4.1.1

a. "CONS" b. "CONS" c. "cons" d. "CONS" e. "CONS" f. "CoNS"

4.1.2 a. T b. NIL c. T d. T e. NIL

4.1.3 The value of the global variable *print-case* in this book is different from the one in *Introduction to Common Lisp*. In *Introduction to Common Lisp*, the value of *print-case* is set to :downcase, and the value in this book is set to :upcase. Also, *print-case* can be set to :capitalize (see Section 11.2 in *Introduction to Common Lisp*).

4.2.1 KCL has the following list of packages when it is started:

```
>(list-all-packages)
(#<"COMPILER" package>
 #<"SYSTEM" package>
 #<"KEYWORD" package>
 #<"USER" package>
 #<"LISP" package>)
```

Besides the four packages which are mentioned in *Introduction to Common Lisp*, the package called compiler is also listed here. The compiler package contains the symbols used by the KCL compiler.

4.3.1

```
(let ((x nil))
  (do-external-symbols (s (find-package 'lisp) x)
    (when (fboundp s) (push s x))))
```

Some systems register those symbols which are not defined by Common Lisp as external symbols of the lisp package. In this case the value of the above expression may include more symbols than you expect.

4.3.2

```
(let ((x nil))
  (do-external-symbols (s (find-package 'lisp) x)
    (when (boundp s) (push s x))))
```

For the same reason as discussed in the answer to Exercise 4.3.1, the value of this expression may include more symbols than you expect.

4.3.3

```
(let ((x nil))
  (do-external-symbols (s (find-package 'lisp) x)
    (when (constantp s) (push s x))))
```

Once again the value of this expression may include more symbols than you expect.

4.3.4 The value of the `let` expression would be either `(X S)` or `(S X)`. When the `fresh` package is created by `in-package`, the package does not include any symbols. When the system reads the `let` expression, it automatically registers the two symbols, `x` and `s`, which appear in the `let` expression, in the current package, `fresh`. Therefore, when `do-symbols` is executed, the symbols `x` and `s` will be bound to the variable `s` either in the order `x`, `s` or in the order `s`, `x`.

If you omit the ⟨⟨package expression⟩⟩ part from the `do-symbols` expression as in this exercise, the system assumes that the current package has been specified implicitly. In this case, since there is no place where you can specify the value returned by the `do-symbols`, the system returns `nil`. In the exercise, we inserted a variable `x` after the `do-symbol` expression since we wanted to make the value of the variable `x` the value of the `let` expression.

4.4.1

```
....

VATICAN>john
[23]

VATICAN>stanford::john
[42]

......

STANFORD>john
[42]

STANFORD>vatican::john
[23]

......
```

```
>stanford::john
[42]


>vatican::john
[23]


>(symbol-name 'stanford::john)
["JOHN"]


>(symbol-name 'vatican::john)
["JOHN"]
```

4.4.2

```
(defun my-symbol (symbol)
  (intern (concatenate 'string
                       "MY-"
                       (symbol-name symbol))))
```

Since `intern` returns two values, this function also returns two values. If you want to force the function to return only the first value, you can use `values` as follows:

```
(defun my-symbol (symbol)
  (values (intern (concatenate 'string
                               "MY-"
                               (symbol-name symbol)))))
```

4.4.3

```
(defun explode (s)
  (let ((pname (symbol-name s))
        (x nil))
    (do ((i (1- (length pname)) (1- i)))
        ((minusp i) x)
      (push (intern (subseq pname i (1+ i)))
            x))))
```

4.4.4

```
(defun implode (x)
  (let ((pname ""))
    (dolist (s x (values (intern pname)))
      (setq pname
            (concatenate 'string
                         pname
                         (symbol-name s))))))
```

The above definition does not execute efficiently because it creates a new string each time it calls `concatenate`. A more efficient implementation would use `apply` (see Section 6.3 in *Introduction to Common Lisp*) and `mapcar` (see Section 8.2 in *Introduction to Common Lisp*).

```
(defun implode (x)
  (values
    (intern
      (apply #'concatenate
             'string
             (mapcar #'symbol-name x)))))
```

4.5.1

a. `T` b. `T` c. `:TEST` d. `:TEST` e. `"TEST"`

4.6.1

```
......

>(car '(a b c))
[(CAR A B C)]

>(lisp:car '(a b c))
[A]

......

>(cdr '(a b c))
[A]

>(lisp:cdr '(a b c))
[(B C)]
```

4.7.1

```
(defun expand-or (form)
  (if (endp (cdr form))
      nil
      (expand-or-1 (cdr form))))

(defun expand-or-1 (forms)
  (if (endp (cdr forms))
      (car forms)
      (let ((temp (gensym)))
        (list 'let
              (list (list temp (car forms)))
              (list 'if temp
                        temp
                        (expand-or-1 (cdr forms)))))))
```

The definition of the auxiliary function `expand-or-1` is complicated, but it can be made simpler by using a backquote macro (see Chapter 7 of *Introduction to Common Lisp*).

```
(defun expand-or-1 (forms)
  (if (endp (cdr forms))
      (car forms)
      (let ((temp (gensym)))
        `(let ((,temp ,(car forms)))
           (if ,temp
               ,temp
               ,(expand-or-1 (cdr forms)))))))
```

5 List Structure

A list is represented as a combination of cons cells. The functions `car` and `cdr` pick out the car part of a cons cell and the cdr part of a cons cell, respectively. `cons` makes a new cell and sets its car part and cdr part. `rplaca` and `rplacd` are used to change the content of an existing cell.

In addition to lists, dotted pairs and dotted lists are data structures built out of combinations of cons cells. These are printed using a dot ".".

The `eq` function is used to check whether two data structures are the same. `eql` is used to check whether two data structures are the same, or whether their values are the same number, or, if they are character data, whether they represent the same letter. Common Lisp uses this `eql` as the standard way to compare two data. `equal` checks for the equality of the content of the data structure. Data that are printed in the same way are usually equal in the `equal` sense.

- summary of functions

  (`consp` ⟨data⟩)
  (`listp` ⟨data⟩)
  (`atom` ⟨data⟩)

  (`rplaca` ⟨cons⟩ ⟨data⟩)
  (`rplacd` ⟨cons⟩ ⟨data⟩)

  (`list*` ⟨data⟩ ... ⟨data⟩ ⟨data⟩)
  (`append` ⟨list⟩ ... ⟨list⟩)
  (`nconc` ⟨list⟩ ... ⟨list⟩)

  (`eq` ⟨data⟩ ⟨data⟩)
  (`eql` ⟨data⟩ ⟨data⟩)
  (`equal` ⟨data⟩ ⟨data⟩)

5-1 cons Cells

5.1.1 Illustrate the following data using cons cells:

a. `(good luck)`
b. `(1 2 3)`
c. `(10 - 3 = 7)`
d. `(setq x y)`
e. `(- (* b b) (* 4 c))`

5.1.2 Fill in [ ] in the following example:

```
>(setq x (cons 'v 10))
[ ]

>(car x)
[ ]

>(cdr x)
[ ]

>(list x x x)
[ ]

>(cons x x)
[ ]
```

5.1.3 Fill in [ ] in the following example:

```
>'(a . b)
[ ]

>'(a . nil)
[ ]

>'(a . (b . (c . nil)))
[ ]

>'(a . (b . (c . d)))
[ ]

>'(a . (b c d))
[ ]
```

5.1.4 What are the values of the following expressions:

a. `(consp 'a)`
b. `(consp '(a . b))`
c. `(consp (cdr '(a . b)))`
d. `(consp (cdr '(a b)))`
e. `(consp (cddr '(a b)))`
f. `(consp '(a b c . d))`

5.1.5 What are the values of the following expressions:

a. `(listp 'a)`
b. `(listp '(a . b))`
c. `(listp (cdr '(a . b)))`
d. `(listp (cdr '(a b)))`
e. `(listp (cddr '(a b)))`
f. `(listp '(a b c . d))`

5.1.6 What are the values of the following expressions:

a. `(atom 'a)`
b. `(atom '(a . b))`
c. `(atom (cdr '(a . b)))`
d. `(atom (cdr '(a b)))`
e. `(atom (cddr '(a b)))`
f. `(atom '(a b c . d))`

5.1.7 Recursive data structure

A list can be defined as

1. the empty list (`nil`) is a list;
2. a cons datum if its `cdr` part is a list;
3. anything other than the above is not a list.

Notice that a list is defined using a word "list" as in "cdr part is a list." Such a definition is called a **recursive definition**, and data structures which can be defined recursively is called a **recursive data structure**. Lists are recursive data structures.

It is natural for a function which takes recursively defined data to be defined recursively. For example, the predicate which checks if its argument is a list can be defined recursively as follows:

```
(defun true-list-p (x)
  (or (null x)
      (and (consp x)
           (true-list-p (cdr x)))))
```

(Unfortunately, this definition does not work when a circular list—see Section 5.2 in *Introduction to Common Lisp*—is given as an argument.)

A dotted pair is defined as "a cons whose cdr part is not the empty list nor a cons." The predicate which checks if some datum is a dotted pair or not can be defined as follows:

```
(defun dotted-pair-p (x)
  (and (consp x)
       (not (or (null (cdr x))
                (consp (cdr x))))))
```

A dotted list has a recursive structure just like lists. Define a dotted list recursively using the definition of the list shown above. Also, define the predicate called `dotted-list-p` which checks whether the given data is a dotted list or not.

```
>(dotted-list-p 'a)
NIL

>(dotted-list-p '(a b))
NIL

>(dotted-list-p '(a . b))
NIL

>(dotted-list-p '(a b . c))
T
```

5.1.8 Determine how many cons cells are used to make each of the following data structures:

a. `(a b c)`
b. `(a . b)`
c. `(a b c d . e)`
d. `((a b) (c) (d e f))`

5.1.9 Define a function `count-cons` which, when it is given a data structure made of only cons cells and symbols, determines the number of cons cells contained in that structure.

```
>(count-cons '(a (b c) d))
5
```

5-2 Destructive List Operations

5.2.1 Fill in [ ] in the following example:

```
>(setq x '(a b c))
(A B C)

>(rplaca (cdr x) 'bb)
(BB C)

>x
[ ]

>(rplaca (cddr x) 'cc)
(CC)

>x
[ ]

>(rplacd (cddr x) 'dd)
(CC . DD)

>x
[ ]
```

5.2.2 The system function nconc can take an arbitrary number of arguments. Define the function nconc2 which takes exactly two arguments and behaves just like nconc.

```
>(setq x '(a b))
(A B)

>(nconc2 x '(c d))
(A B C D)

>x
(A B C D)

>(nconc2 x '(e f))
(A B C D E F)

>x
(A B C D E F)
```

```
>(setq y nil)
NIL

>(nconc2 y '(1 2))
(1 2)

>y
NIL
```

5.2.3 Sorting

Let's think about defining a function **sort-numbers** which, when it is given a list of numbers, returns a new list of those numbers in increasing order.

```
>(sort-numbers '(7 2 0 9))
(0 2 7 9)
```

There are many methods for **sorting** lists. Here, we will use one of the simplest. First, we define the function.

```
(defun insert-number (n numbers)
  (cond ((null numbers)
         (list n))
        ((<= n (car numbers))
         (cons n numbers))
        (t
         (cons (car numbers)
               (insert-number n (cdr numbers))))))
```

The above function inserts the number **n** into the appropriate place in a list of numbers called **numbers**, which has already been sorted.

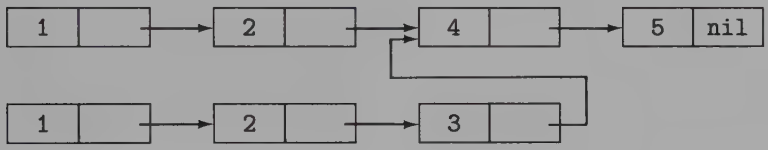
```
>(insert-number 3 '(1 2 4 5))
(1 2 3 4 5)
```

Using this function, **sort-numbers** can be defined as

```
(defun sort-numbers (numbers)
  (if (null numbers)
      nil
      (insert-number
        (car numbers)
        (sort-numbers (cdr numbers)))))
```

In the above definition of `insert-number`, the system copies parts of the original list of numbers every time a number is inserted. In the above example, when 3 is inserted into

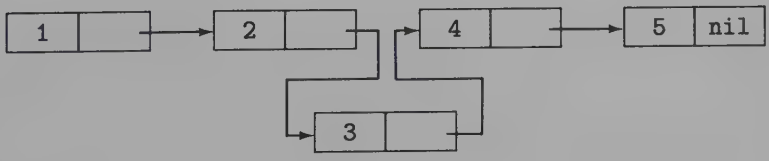
copies are made of the cell for 1 and the cell for 2.



This is wasteful. It takes some time for the system to make the new cons cells and the two cells which are copied become "garbage" (see Section 5.4 in *Introduction to Common Lisp*). When the list to be sorted is short as in this example, there might only be a little waste, but when the list is made of 1000 elements, in the worst case the following amount of garbage will be created:

```
>(let ((n 0))
   (dotimes (i 1000 n)
     (incf n i)))
499500
```

In order not to use up so many cells, you should make the cdr part of the cell, which is right in front of the cell, point to the cell to be inserted.



Redefine `insert-number` using this method.

5-3 Equality

5.3.1 What are the values of the following expressions:

a.
```
(eq 'a 'a)
```
b.
```
(eq 'a 'b)
```
c.
```
(let ((x 'a))
  (eq x 'a))
```
d.
```
(eq '(a b) '(a b))
```
e.
```
(let ((x '(a b)))
  (eq x x))
```
f.
```
(let ((x '(a b)))
  (eq x '(a b)))
```
g.
```
(let* ((x '(a b))
       (y x))
  (eq x y))
```

5.3.2 In the previous exercise, replace `eq` by `eql` and then determine the values.

5.3.3 In the previous exercise, replace `eq` by `equal` and then determine the values.

5.3.4 Binary search tree (1)
One popular data structure for searching data is called a **binary search tree**. A binary search tree is generally used for searching for data over which there is a total order. Here, as a special case, we will consider searching for numbers. Figure 5.1 is a sample binary search tree for numbers. Since the branches are spreading downward, this may look like a root rather than a tree. If this really bothers you, look at this figure upside down. The small circles in Figure 5.1 are called **nodes**, and each of them stores one number. The nodes are connected by **branches**. The branches form the **tree structure** as a whole. Only two branches come out from a node. This is why the tree is called a binary tree. Suppose one node contains the number n. Nodes descending in the lefthand direction from this node only contain numbers which are smaller than n. Nodes descending in the righthand direction from this node only contain numbers which are bigger than n. By using this convention the system can search numbers effectively. In order to search for a number n, the system starts from the top node and descends doing the following:

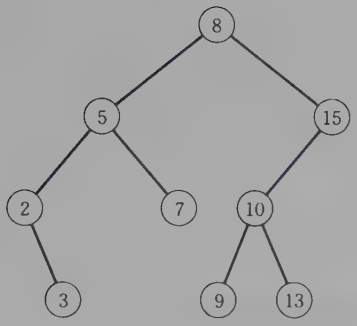


Figure 5.1 Example of binary search tree

1. If the number of the node is the same as n, the system ends the search.
2. If the number of the node is smaller than n, it goes to the left branch. If there is no left branch, it ends the search.
3. If the number of the node is bigger than n, it goes to the right branch. If there is no right branch, it ends the search.

Step 1 indicates that the system finds the number it was searching for in the search tree. If the search ends in 2 or 3, it means that the system failed to find the number. For example, to find 9 in Figure 5.1, the system will find it by going down

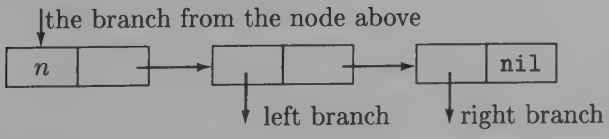
8 → 15 → 10 → 9

To find 1, the system will go down

8 → 5 → 2

Since there is no branch to the left at the end, the tree does not contain 1.

A binary search tree can be expressed in Lisp using a cons cell for each node as follows:



Here, n expresses the number saved in this node. If there is no left branch or right branch, you should put `nil` instead of a pointer to the next cell.

Create the binary search tree in Figure 5.1 as a Lisp data structure like the above and assign it to the global variable called `tree`. Then, define a function `bs-tree-search` which searches for a number using this tree. This function should return `t` if it finds the number and `nil` when it cannot find the number.

```
>(bs-tree-search 9 bs-tree)
T

>(bs-tree-search 1 bs-tree)
NIL
```

5.3.5 Binary search tree (2)

Define a function `bs-tree-insert` which takes a number and a binary search tree and, if it cannot find the number in the binary search tree, adds a node at an appropriate place in the tree where the number can be stored. The tree should be constructed as described in Exercise 5.3.4, and this function should return the same value as `bs-tree-search`.

```
>(bs-tree-search 16 bs-tree)
NIL

>(bs-tree-insert 16 bs-tree)
NIL

>(bs-tree-search 16 bs-tree)
T
```

5.3.6 Game trees (1)

A simple game called "the stone picking up game" is played by two people. Suppose the number of stones is n at the start. Two people take turns by picking up at most three stones at a time. The person who picks up the last stone will lose.

Figure 5.2 illustrates the part of the **game tree** in which A and B are two players, there are initially sixteen stones, and A plays first. In general, each node of a game tree expresses the possible situation and the branches correspond to the next possible move.

In this example, "situation" means the number of stones left, and "next move" means the number of stones to be picked up. Each node has as many branches as there are as possible next moves and each branch connects to the next situation. Since there is not enough space to show the whole game tree, we will show you the bottom part of the leftmost branch of the game tree in Figure 5.3.

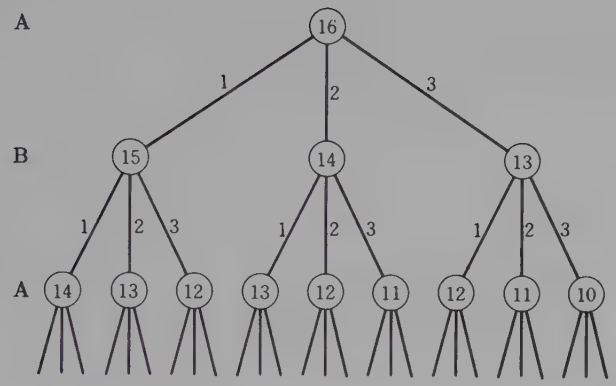


Figure 5.2 Game tree of the stone picking up game (1)

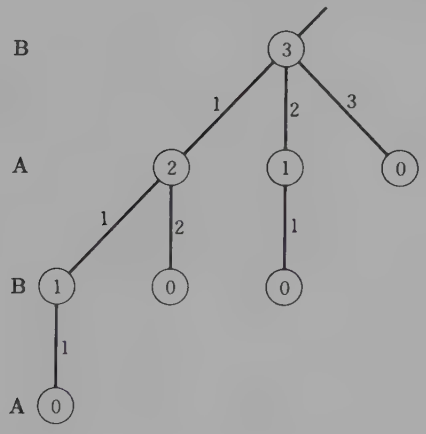


Figure 5.3 Game tree of the stone picking game (2)

Suppose we want to express this game tree using the following lists:

(⟪number of stones⟫ ⟪situation when one stone is picked⟫
⟪situation when two stones are picked⟫
⟪situation when three stones are picked⟫)

When the number of the stones to be picked is 2, 1, or 0, you want to use the following:

```
(2 ⟪situation when one stone is picked⟫
   ⟪situation when two stones are picked⟫ )
(1 ⟪situation when one stone is picked⟫ )

(0)
```

For example, the game tree shown in Figure 5.2 can be expressed as

```
(16 (15 (14 ( ... (3 (2 (1 (0))
                        (0))
                    (1 (0))
                    (0))
                ... )
            ... )
        (13 ... )
        (12 ... ))
    (14 (13 ... )
        (12 ... )
        (11 ... ))
    (13 (12 ... )
        (11 ... )
        (10 ... )))
```

Define a function called `nim` which displays the game tree like the one shown above when the initial number of stones is given.

5-4 Garbage Collection

Most Common Lisp systems have the ability to display information on a garbage collection and to let a user explicitly start a garbage collection. Usually, however, it is started automatically by the system when it is necessary. In KCL, if you set the value of a variable called `*notify-gbc*` to `t`, the system notifies you when a garbage collection starts and when it ends.

```
>system:*notify-gbc*
NIL

>(dotimes (i 50 'done) (make-list 1000))
DONE
```

```
>(setq system:*notify-gbc* t)
T

>(dotimes (i 50 'done) (make-list 1000))

GBC invoked
GBC finished

GBC invoked
GBC finished
DONE

>
```

This example tells you there were two garbage collections during the evaluation of the latter `dotimes` expression. Also, in KCL, as soon as you type

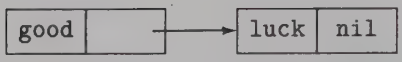
```
>(gbc nil)
```

the system starts a garbage collection. Check how long one garbage collection takes on your Common Lisp system.

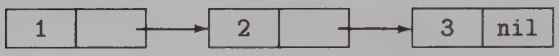
Answers to Exercises

5.1.1

a.



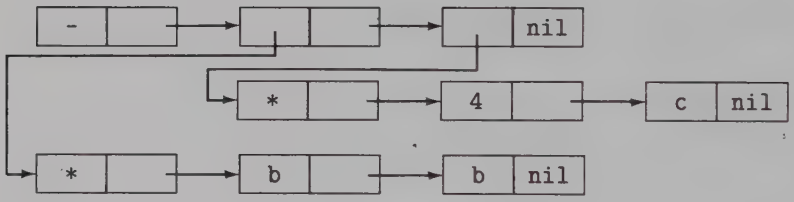
b.



c.

d.

e.



5.1.2

```
>(setq x (cons 'v 10))
[(V . 10)]

>(car x)
[V]

>(cdr x)
[10]

>(list x x x)
[((V . 10) (V . 10) (V . 10))]

>(cons x x)
[((V . 10) V . 10)]
```

5.1.3

```
>'(a . b)
[(A . B)]

>'(a . nil)
[(A)]

>'(a . (b . (c . nil)))
[(A B C)]

>'(a . (b . (c . d)))
[(A B C . D)]

>'(a . (b c d))
[(A B C D)]
```

5.1.4 a. NIL b. T c. NIL d. T e. NIL f. T

5.1.5 a. NIL b. T c. NIL d. T e. T f. T

5.1.6 a. T b. NIL c. T d. NIL e. T f. NIL

5.1.7

1. a cons cell whose cdr part is a dotted pair is a dotted list.
2. a cons cell whose cdr part is a dotted list is a dotted list.
3. anything else is not a dotted list.

1 and 2 can be summarized as

> a cons cell whose cdr part is either a dotted pair or a dotted list is a dotted list.

The function `dotted-list-p` can be defined as follows using the function `dotted-pair-p` defined in the Exercise 5.1.7.

```
(defun dotted-list-p (x)
  (and (consp x)
       (or (dotted-pair-p (cdr x))
           (dotted-list-p (cdr x)))))
```

5.1.8 a. 3 b. 1 c. 4 d. 9

You should look at the data as the value of an expression that calls `cons`. For example, the last line

```
((a b) (c) (d e f))
```

can be obtained by evaluating

```
(cons (cons 'a (cons 'b nil))
      (cons (cons 'c nil)
            (cons (cons 'd
                        (cons 'e
                              (cons 'f nil)))
                  nil)))
```

Since the system will use one cons cell each time it calls `cons`, nine cons cells in total will be used.

5.1.9

```
(defun count-cons (x)
  (if (symbolp x)
      0
      (+ 1
         (count-cons (car x))
         (count-cons (cdr x)))))
```

5.2.1

```
>(setq x '(a b c))
(A B C)

>(rplaca (cdr x) 'bb)
(BB C)

>x
[(A BB C)]

>(rplaca (cddr x) 'cc)
(CC)

>x
[(A BB CC)]
```

```
>(rplacd (cddr x) 'dd)
(CC . DD)

>x
[(A BB CC . DD)]
```

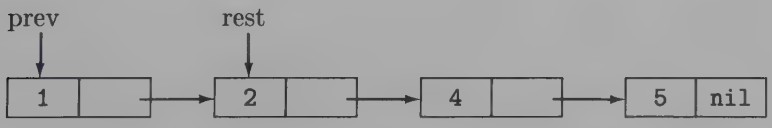
5.2.2

```
(defun nconc2 (x y)
  (if (endp x)
      y
      (do ((z x (cdr z)))
          ((endp (cdr z)) (rplacd z y) x))))
```

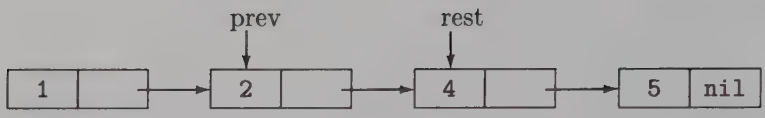
5.2.3

```
(defun insert-number (n numbers)
  (cond ((null numbers)
         (list n))
        ((<= n (car numbers))
         (cons n numbers))
        (t (do ((prev numbers rest)
                (rest (cdr numbers)
                      (cdr rest)))
               ((or (endp rest)
                    (<= n (car rest)))
                (rplacd prev (cons n rest))))
           numbers)))
```

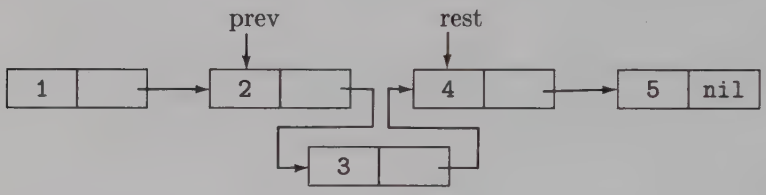
Let's look at how the values of the two local variables of the `do` expression, `prev` and `rest`, change in the example shown in the exercise. The initial situation is as follows:



At this point, neither `(endp rest)` nor `(<= n (car rest))` is true. Since the `do` expression does not have a main body, the system moves to the next step, producing the following:



Here, since `(<= n (car rest))` is true, even though `(endp rest)` is still not true, the system does not repeat the `do` expression any more. The form `(cons n rest)` creates a cons cell whose car part is 3 and cdr part is `rest`, and `rplacd` replaces the cdr part of the `prev` cell with this new cell.



This form of `insert-number` operates destructively on lists, but when the number is inserted at the beginning of the list it will not affect the original list since `rplacd` will not be used.

```
>(setq x '(1 2 4 5))
(1 2 4 5)

>(insert-number 3 x)
(1 2 3 4 5)

>x
(1 2 3 4 5)

>(insert-number 0 x)
(0 1 2 3 4 5)

>x
(1 2 3 4 5)
```

5.3.1 a. T b. NIL c. T d. NIL e. T f. NIL g. T

5.3.2 a. T b. NIL c. T d. NIL e. T f. NIL g. T

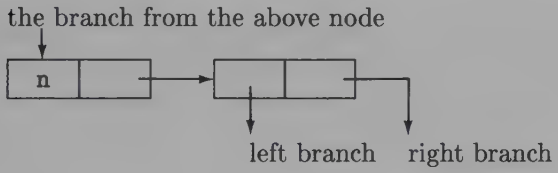
5.3.3 a. T b. NIL c. T d. T e. T f. T g. T

5.3.4

```
(setq bs-tree
      '(8 (5 (2 nil (3 nil nil))
             (7 nil nil))
          (15 (10 (9 nil nil)
                  (13 nil nil))
              nil)))

(defun bs-tree-search (n bst)
  (cond ((null bst) nil)
        ((= n (car bst)) t)
        ((< n (car bst))
         (bs-tree-search n (cadr bst)))
        (t (bs-tree-search n (caddr bst)))))
```

This can also be expressed as follows using two cells:



5.3.5

```
(defun bs-tree-insert (n bst)
  (cond ((= n (car bst)) t)
        ((< n (car bst))
         (cond ((null (cadr bst))
                (rplaca (cdr bst)
                        (list n nil nil))
                nil)
               (t (bs-tree-insert n (cadr bst)))))
        (t
         (cond ((null (caddr bst))
                (rplaca (cddr bst)
                        (list n nil nil))
                nil)
               (t (bs-tree-insert n (caddr bst)))))))
```

Here, `(cdr bst)` in

```
(rplaca (cdr bst) (list n nil nil))
```

should not be replaced by `(cadr bst)` since this will replace the car part of `nil`.

5.3.6 One simple definition is

```
(defun nim (n)
  (case n
    (0 (list 0))
    (1 (list 1 (nim 0)))
    (2 (list 2 (nim 1) (nim 0)))
    (otherwise
       (list n
             (nim (- n 1))
             (nim (- n 2))
             (nim (- n 3))))))
```

Unfortunately this is not efficient because so many duplicate calculations are involved. If you examine the game tree carefully, you can see that the game tree for n stones can easily be constructed from the game tree for $n-1$ stones. Therefore, the following definition of this function will not make any duplicate calculations:

```
(defun nim (n)
  (case n
    (0 (list 0))
    (1 (list 1 (nim 0)))
    (2 (list 2 (nim 1) (nim 0)))
    (otherwise
      (let ((x (nim (1- n))))
        (list n
              x
              (cadr x)
              (caddr x))))))
```

5.4.1 The time required for one garbage collection is different on different systems and it is also affected by the data used. Here we show you an example using KCL on the author's system for your reference:

```
>(setq system:*notify-gbc* t)
T
```

```
>(time (gbc nil))

GBC invoked
GBC finished
real time : 1.000 secs
run time  : 0.586 secs
NIL
```

In this example, it takes 0.586 seconds for the garbage collection.

6 Functions

Functions have four different kinds of parameters:

Required parameters
Optional parameters
Rest parameters
Keyword parameters

You can specify an expression for the default value of an optional parameter or a keyword parameter. You can use the supplied-p variable to determine whether an argument is given or not. Such specifications are done using lambda lists.

While functions defined using `defun` are global, functions defined using `flet` and `labels` forms are local.

Using `funcall` and `apply`, the system can decide which function to call dynamically during execution. In this case, one of the following forms is used to specify the function to be used:

A symbol which is the name of a function
A lambda expression
A function definition itself
Closures

A `function` form or its abbreviation using "`#'`" is used in order to specify the function definition itself. Such expressions are also used in making closures.

- summary of functions

```
(flet ((⟪function name⟫ ⟪lambda list⟫
                ⟪expression⟫...⟪expression⟫ )
        ...
        (⟪function name⟫ ⟪lambda list⟫
                ⟪expression⟫...⟪expression⟫ ))
 ⟪expression⟫...⟪expression⟫ )
(labels (( ⟪function name⟫ ⟪lambda list⟫
                ⟪expression⟫...⟪expression⟫ )
          ...
          ( ⟪function name⟫ ⟪lambda list⟫
                ⟪expression⟫...⟪expression⟫ ))
  ⟪expression⟫...⟪expression⟫ )

(funcall ⟨function⟩ ⟨argument⟩...⟨argument⟩ )
(apply ⟨function⟩ ⟨argument⟩...⟨argument⟩ ⟨argument list⟩)

(function ⟪lambda expression or symbol⟫ )
```

6-1 Lambda Expressions

6.1.1 What are the values of the following expressions:

a.
```
((lambda (a b) (list a b))
   1 2)
```

b.
```
((lambda (a b &optional (c 30))
   (list a b c))
 1 2)
```

c.
```
((lambda (a b &optional (c 30))
   (list a b c))
 1 2 3)
```

d.
```
((lambda (a b &optional (c (- a b)) d)
   (list a b c d))
 1 2)
```

e.
```
((lambda (a b &optional (c (- a b)) d)
   (list a b c d))
 1 2 3)
```

f.
```
((lambda (a b &optional (c (- a b)) d)
   (list a b c d))
 1 2 3 4)
```

```
g. ((lambda (a b &optional (c 30) &rest d)
      (list a b c d))
    1 2)
h. ((lambda (a b &optional (c 30) &rest d)
      (list a b c d))
    1 2 3)
i. ((lambda (a b &optional (c 30) &rest d)
      (list a b c d))
    1 2 3 4)
j. ((lambda (a b &optional (c 30) &rest d)
      (list a b c d))
    1 2 3 4 5)
```

6.1.2 The lambda list of a `defun` form determines the number of arguments that a function can take. For example, the function defined as

```
(defun foo (a &optional (b 1)) ...)
```

needs at least one argument since there is one required parameter. Since there is one optional parameter, it can take one more argument. Therefore, this function can take either one or two arguments. When the number of arguments is not one or two, the result depends on the version of Common Lisp that you are using and may be different even in the same system depending on the situation. In KCL the following happens:

```
>(defun foo (a &optional (b 1)) (list a b))
FOO

>(foo 1)
(1 1)

>(foo 1 2)
(1 2)

>(foo)
Error: FOO requires more than zero arguments.
Error signalled by FOO.

Broken at FOO.  Type :H for Help.
>>:q
```

```
Top level.
>(foo 1 2 3)
Error: FOO requires less than three arguments.
Error signalled by FOO.

Broken at FOO.  Type :H for Help.
>>:q

Top level.
>
```

State the range of the number of the arguments which each of the following function can take. Each function is defined as follows using the **defun** expression:

a. `(defun fun1 () ...)`
b. `(defun fun2 (a b) ...)`
c. `(defun fun3 (a b &optional (c 1) (d 2)) ...)`
d.
```
(defun fun4 (a b &optional (c 1)
                 &rest d)
...)
```
e. `(defun fun5 (a &rest b) ...)`
f. `(defun fun6 (&rest a) ...)`
g.
```
(defun fun7 (&optional (a 1)
             &aux (b 2) (c 3))
...)
```

6.1.3 The function **intern** (see Section 4.4 of *Introduction to Common Lisp*) takes two arguments and is called

(**intern** ⟨character string⟩ ⟨package⟩)

The second argument, ⟨package⟩, can be omitted. In such cases the value of ***package*** is used as the default value. Show the lambda list used in the definition of the function **intern**.

6.1.4 Suppose we define a function **exchange** for computing the changing of dollars to yen. Suppose the global variable ***current-rate*** contains the current exchange rate (yen/dollar). If one dollar is 134 yen, you can set

```
>(setq *current-rate* 134)
```

The first argument to the function **exchange** should be the amount of dollars. If there is only one argument, the function should use the current

exchange rate. If there are two arguments, the function should use the number passed as the second argument for the exchange rate. For example,

```
>(setq *current-rate* 134)
134

>(exchange 100)
13400

>(exchange 100 360)
36000

>(setq *current-rate* 360)
360

>(exchange 100)
36000
```

Define the function exchange.

6-2 Local Functions

6.2.1 Find the value of the following expression:

```
>(flet ((cons (x y) (cons y x))
        (car (x) (cdr x))
        (cdr (x) (car x)))
   (let ((x '(a . b)))
     (list (car x)
           (cdr x)
           (cons (car x) (cdr x)))))
```

6.2.2 In Exercise 3.1.7 we learned to define, in self-tail-recursive style, the function `list-sum`, which computes the sum of numbers in a numerical list by using an auxiliary function.

```
(defun list-sum (x) (list-sum-aux x 0))
```

```
(defun list-sum-aux (x so-far)
  (if (null x)
      so-far
      (list-sum-aux (cdr x)
      (+ (car x) so far))))
```

Redefine the function `list-sum` so that the auxiliary function `list-sum-aux` becomes a local function of `list-sum`.

6.2.3 Polynomial expressions (1)

Suppose we represent the single variable polynomial expression

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

using the numerical list

$$(a_0\ a_1\ a_2\ \ldots a_n)$$

Define a function `poly-dif` which computes the derivative

$$f'(x) = a_1 + 2a_2 x + \cdots + n a_n x^{n-1}$$

of the given polynomial. For example, the derivative of

$$f(x) = 1 + 2x + 3x^2$$

is

$$f'(x) = 2 + 6x$$

Therefore,

```
>(poly-diff '(1 2 3))
(2 6)
```

6.2.4 Polynomial expressions (2)

Define a function `poly-eval` which takes a single variable polynomial whose coefficients are numbers and the value to be assigned to the variable and returns the value of the polynomial. Polynomials should be represented as in Exercise 6.2.3. For example, in the case of

$$f(x) = 1 + 2x + 3x^2$$

we have

$$f(2) = 1 + 2 \times 2 + 3 \times 2^2 = 17$$

Therefore

```
>(poly-eval '(1 2 3) 2)
7
```

6-3 Functions for Function Calls

6.3.1 What are the values of the following expressions:

a. `(funcall 'list 1 (- 8 6))`
b. `(funcall (cadr '(cons list)) 1 2)`
c. `(funcall (intern "LIST") 1 2)`
d. `(apply 'list 1 (cons 2 nil))`
e. `(apply 'list '(1 2))`
f. `(apply 'list 1 2 nil)`
g. `(apply 'list (funcall 'list 1 2))`

6.3.2 The function **append2**, which takes two arguments and which has the same values as **append**, is defined in Exercise 3.1.4. Define the function **my-append**, which is the same as **append**, using **append2** as a local function. For example,

```
>(my-append)
NIL

>(my-append '(a b))
(A B)

>(my-append '(a b) '(c d))
(A B C D)

>(my-append '(a b) '(c d) '(e f))
(A B C D E F)
```

6.3.3 Write an expression which has the same value as

`(multiple-value-list` ⟪expression⟫ `)`

using `multiple-value-call`.

6.3.4 Interpreter (1)
Let's define a function `simple-eval` which is an abbreviated version of the system function `eval`. Suppose `simple-eval` evaluates only forms which consist of either numbers or function calls starting with function names.

```
>(simple-eval '1)
1
```

```
>(simple-eval '(- (* 2 4) 3))
5

>(trace simple-eval)
(SIMPLE-EVAL)

>(simple-eval '(- (* 2 4) 3))
  1> (SIMPLE-EVAL (- (* 2 4) 3))
    2> (SIMPLE-EVAL (* 2 4))
      3> (SIMPLE-EVAL 2)
      <3 (SIMPLE-EVAL 2)
      3> (SIMPLE-EVAL 4)
      <3 (SIMPLE-EVAL 4)
    <2 (SIMPLE-EVAL 8)
    2> (SIMPLE-EVAL 3)
    <2 (SIMPLE-EVAL 3)
  <1 (SIMPLE-EVAL 5)
 5
```

It will be easier to compute the arguments for a function call if we first define the function **simple-evlis** which takes a list of forms and returns a list of the values of the forms in the list.

```
>(simple-evlis '(10 (cons 1 2)))
(10 (1 . 2))

>(simple-evlis '())
NIL
```

Since you will be asked in the exercises below to enlarge the kinds of forms which **simple-eval** will evaluate, we would like to introduce some relevant terminology. First, we will use SF for the set of simple forms evaluated by **simple-eval**. In this exercise, SF means

⟪numbers⟫

or

(⟪function name⟫ ⟪SF⟫ ... ⟪SF⟫)

We will write this as

⟪SF⟫ ::= ⟪number⟫
| (⟪function name⟫ ⟪SF⟫ ... ⟪SF⟫)

"::=" is read "means" and "|" is read "or."

6-4 Lexical Closures

6.4.1 **funargs** (1)

Fill in [ ] in the following interaction:

```
>(setq x 1)
1

>(defun foo (f x) (funcall f x))
FOO

>(let ((x 10))
   (foo '(lambda (y) (+ x y)) 100))
[ ]

>(let ((x 10))
   (foo #'(lambda (y) (+ x y)) 100))
[ ]
```

6.4.2 **funargs** (2)

Fill in [ ] in the following interaction:

```
>(setq x 1)
1

>(let ((x 100))
   (setq f #'(lambda () (incf x)))
   (setq g '(lambda () (incf x))))
(LAMBDA () (INCF X))

>(funcall f)
[ ]

>(funcall g)
[ ]

>(funcall f)
[ ]

>x
[ ]
```

Answers to Exercises

6.1.1

a. `(1 2)`
b. `(1 2 30)`
c. `(1 2 3)`
d. `(1 2 -1 NIL)`
e. `(1 2 3 NIL)`
f. `(1 2 3 4)`
g. `(1 2 30 NIL)`
h. `(1 2 3 NIL)`
i. `(1 2 3 (4))`
j. `(1 2 3 (4 5)))`

6.1.2

a. 0 (no arguments)
b. 2
c. 2, 3, or 4
d. more than 2
e. more than 1
f. more than 0 (any number)
g. 0 or 1

6.1.3

```
(defun intern (string
               &optional (package *package*))
  ... )
```

6.1.4

```
(defun exchange (dollars
                 &optional
                  (rate *current-rate*))
  (* dollars rate))
```

6.2.1

```
(B A (A . B))
```

6.2.2 Since `list-sum-aux` is defined recursively, you should use `labels` rather than `flet` in the definition of `list-sum`. This is done by inserting the part after `defun`, in the original definition of `list-sum-aux`, at the beginning of the `labels` expression.

```
(defun list-sum (x)
  (labels ((list-sum-aux (x so-far)
             (if (null x)
                 so-far
                 (list-sum-aux
                   (cdr x)
                   (+ (car x) so-far)))))
    (list-sum-aux x 0)))
```

You can define `list-sum` as follows by placing the parameter `so-far` outside the auxiliary function:

```
(defun list-sum (x &aux (so-far 0))
  (labels ((list-sum-aux (x)
             (if (null x)
                 so-far
                 (progn
                   (incf so-far (car x))
                   (list-sum-aux (cdr x))))))
    (list-sum-aux x)))
```

You can also place the parameter `x` outside `labels`:

```
(defun list-sum (x &aux (so-far 0))
  (labels ((list-sum-aux ()
             (if (null x)
                 so-far
                 (progn
                   (incf so-far (car x))
                   (pop x)
                   (list-sum-aux)))))
    (list-sum-aux)))
```

Since, in the above definition, the auxiliary function `list-sum-aux` is only used for the side effect of updating the value of `so-far`, you can also define it as

```
(defun list-sum (x &aux (so-far 0))
  (labels ((list-sum-aux ()
             (unless (null x)
               (incf so-far (car x))
               (pop x)
               (list-sum-aux))))
    (list-sum-aux)
    so-far))
```

From the point of view of readability, the first definition would be the best.

6.2.3

```
(defun poly-diff (poly)
  (labels ((poly-diff1 (poly power)
             (if (endp poly)
                 nil
                 (cons (* power (car poly))
                       (poly-diff1
                         (cdr poly)
                         (1+ power))))))
    (poly-diff1 (cdr poly) 1)))
```

6.2.4

```
(defun poly-eval (poly x)
  (labels ((poly-eval1 (poly power-x)
             (if (endp poly)
                 0
                 (+ (* (car poly) power-x)
                    (poly-eval1
                      (cdr poly)
                      (* power-x x))))))
    (poly-eval1 poly 1)))
```

6.3.1 The values are all `(1 2)`.

6.3.2

```
(defun my-append (&rest x)
  (labels ((append2 (x y)
             (if (null x)
                 y
                 (cons (car x)
                       (append2 (cdr x) y)))))
    (if (null x)
        nil
        (append2 (car x)
                 (apply #'my-append
                        (cdr x))))))
```

6.3.3

```
(multiple-value-call #'list ⟪expression⟫ )
```

6.3.4

```
(defun simple-eval (form)
  (cond ((numberp form) form)
        ((and (consp form)
              (symbolp (car form)))
         (apply (car form)
                (simple-evlis (cdr form))))))

(defun simple-evlis (forms)
  (if (null forms)
      nil
      (cons (simple-eval (car forms))
            (simple-evlis (cdr forms)))))
```

6.4.1

```
>(let ((x 10))
   (foo #'(lambda (y) (+ x y)) 100))
[101]

>(let ((x 10))
   (foo #'(lambda (y) (+ x y)) 100))
[110]
```

The variable x in the expression which creates the closure

```
#'(lambda (y) (+ x y))
```

is a local variable which is bound by the `let` expression. On the other hand, the `x` in

```
'(lambda (y) (+ x y))
```

is the global variable `x`. However, since the lambda expression exists within the scope of the local variable `x`, you could mistake `x` for a local variable. This is called **funarg problem** and has caused errors in Lisp programs.

6.4.2

```
>(funcall f)
[101]

>(funcall g)
[2]

>(funcall f)
[102]

>x
[2]
```

The `x` in the expression

```
'(lambda () (incf x))
```

is a global variable. However, it could be mistaken as a local variable which is bound by the `let` expression. This is also a funarg problem. When a lambda expression is passed as an argument to another function as in Exercise 6.4.1, the problem is called the **downward funarg problem**. When the system returns a lambda expression as a value, it is called the **upward funarg problem**.

7 Macros

Macro expansion is done by calling a macro expansion function. The `defmacro` form is used to define macro expansion functions. A parameter specification which reflects the macro form structure is allowed in the lambda list of a `defmacro` form.

Backquote macros are useful for creating simple definitions of macros.

Besides variables, macros such as `setf`, `incf`, and `push` are useful for updating the content of some "places."

- summary of functions

```
(macro-function ⟨symbol⟩)
(special-form-p ⟨symbol⟩)

(setf ⟪place⟫ ⟨value⟩ ... ⟪place⟫ ⟨value⟩)
(psetf ⟪place⟫ ⟨value⟩ ... ⟪place⟫ ⟨value⟩)
(incf ⟪place⟫ [⟨value⟩])
(decf ⟪place⟫ [⟨value⟩])
(push ⟨data⟩ ⟪place⟫ )
(pop ⟪location⟫ )

(defmacro ⟪macro name⟫ ⟪expand lambda list⟫
   ⟪expression⟫ ... ⟪expression⟫ )
(macrolet (( ⟪macro name⟫ ⟪expand lambda list⟫
                  ⟪expression⟫ ... ⟪expression⟫
              ...
             ( ⟪macro name⟫ . ⟪expand lambda list⟫
                  ⟪expression⟫ ... ⟪expression⟫ ))
   ⟪expression⟫ ... ⟪expression⟫ )
```

7-1 Macros

7.1.1 What are the values of the following expressions:

a. `(macroexpand 10)`
b. `(macroexpand 'x)`
c. `(macroexpand '((lambda (x) (1+ x)) 3))`
d. `(macroexpand '(if b 0 1))`
e. `(macroexpand '(+ 1 2))`
f. `(macroexpand '(return 2))`
g. `(macroexpand (list 'return 2))`

7.1.2 Define a function `my-macroexpand` which takes one argument and which has the same value as `macroexpand`.

7.1.3 Check which system macros are interpreted as special forms in your Common Lisp system.

7-2 setf

7.2.1 Suppose the value of variables `x` and `y` are `(A B C D E)` and `(1 2 3 4 5)`, respectively. What are the values of `x` and `y` after the evaluation of each of the following expressions:

a. `(setf (car (cdr x)) (cadr y))`
b. `(setf (cadr x) (cadr y))`
c. `(setf (caddr x) (caddr y))`
d. `(setf (cdr x) (cdr y))`
e. `(setf (cddddr x) (cddddr y))`

7.2.2 The following `setf` expression and the expression for calling the function `rplaca` both replace the car part of the cons cell, which is the value of the variable `x`, with the number `1`. In other words, the two expressions produce the same side effect.

```
(setf (car x) 1)

(rplaca x 1)
```

Although they "have the same side effect," they do not have the same values. The value of the first expression is 1, and the value of the second expression is the new value of x.

Write the function call expression which has the same effect of each **setf** expression in Exercise 7.2.1.

7.2.3 What is the value of x after the evaluation of each of the following expressions when the value of the variable x is 5:

a. `(incf x)`
b. `(incf x 2)`
c. `(incf x -2)`
d. `(incf x x)`
e. `(decf x)`
f. `(decf x 2)`
g. `(decf x -2)`
h. `(decf x x)`

7.2.4 Find the value of x after the evaluation of each expression when the value of x is `(1 2 3 4 5)`:

a. `(incf (cadr x) 100)`
b. `(decf (car x))`
c. `(push 0 x)`
d. `(pop x)`
e. `(progn (push 10 x) (push 20 x))`
f. `(progn (pop x) (pop x) (pop x))`

7.2.5 Find the value of x after the evaluation of each expression when the value of x is `((1 2 3)(10 20)(100))`:

a. `(setf (cdar x) 'brother)`
b. `(incf (cadr (cadr x)))`
c. `(pop x)`
d. `(pop (caddr x))`
e. `(push 30 (cadr x))`
f. `(push '(a b c) (caddr x))`

7.2.6 Binary search tree (3)
Redefine **bs-tree-insert**, which was defined in Exercise 5.3.5, using **setf**.

7-3 Macro Definitions

7.3.1 Fill in [ ] with the appropriate values in the following interaction:

```
>(defmacro num-filter (var)
   (list 'if
         (list 'numberp var)
         var
         0))
NUM-FILTER

>(macroexpand '(num-filter x))
[ ]
[ ]

>(defun foo (a b c)
   (+ (num-filter a)
      (num-filter b)
      (num-filter c)))
FOO

>(foo 1 2 3)
[ ]

>(foo 'a 2 3)
[ ]

>(foo 'a 'b 'c)
[ ]
```

7.3.2 Define `push-var` which is a macro that behaves like **push** but takes only variables as ⟪place⟫. `push-var` is called as follows:

`(push-var` ⟪expression⟫ ⟪variable⟫ `)`

For example,

```
>(macroexpand '(push-var 1 x))
(SETQ X (CONS 1 X))
T

>(macroexpand '(push-var (+ a b) y))
(SETQ Y (CONS (+ A B) Y))
T
```

7.3.3 Define the macro `my-return` which is equivalent to the system macro `return`.

```
>(macroexpand '(my-return 1))
(RETURN-FROM NIL 1)
T

>(macroexpand '(my-return (+ x y)))
(RETURN-FROM NIL (+ X Y))
T

>(macroexpand '(my-return))
(RETURN-FROM NIL NIL)
```

7.3.4 Define the macro `my-unless` which is equivalent to `unless`.

7.3.5 The following definition of `when` (see Section 7.3 of *Introduction to Common Lisp*)

```
(defmacro when (condition . body)
  (list 'if
        condition
        (cons 'progn body)))
```

uses `progn` even when there is only one expression in the body of the `when` expression. That is,

`(when` ⟪condition⟫ ⟪expression⟫ `)`

is macro-expanded as

`(if` ⟪condition⟫ `(progn` ⟪expression⟫ `))`

In this case it could have been expanded as

`(if` ⟪condition⟫ ⟪expression⟫ `)`

without using `progn`. When there is no expression in the body of the `when` form,

`(when` ⟪condition⟫ `)`

simply evaluates ⟪condition⟫ and returns `nil` as a value independent of the result of evaluating `condition`. In this case it could be macro-expanded to

`(progn` ⟪condition⟫ `nil)`

Redefine `when` so that it macro-expands as described above when there is only one expression or no expressions in the body.

7.3.6 Define the macro `my-and` which is equivalent to **and**. Follow the following way in which the **and** expression is expanded (see Section 2.4 in *Introduction to Common Lisp*):

```
>(macroexpand '(and x y z))
(if x (if y z))
t

>(macroexpand '(and x y))
(if x y)
t

>(macroexpand '(and x))
x
t

>(macroexpand '(and))
t
t
```

7-4 Backquote

7.4.1 The following are the expressions using backquotes. Write an expression without backquotes that is equivalent to each of the following:

a. `` `(if ,condition ,(car body)) ``
b. `` `(setq ,var (cons ,form ,var)) ``
c. `` `(if (not ,condition) ,@body) ``

7.4.2 Redefine `num-filter`, which was defined in Exercise 7.3.1, using backquotes.

7.4.3 Redefine `push-var`, which was defined in Exercise 7.3.2, using backquotes.

7.4.4 Redefine `my-return`, which was defined in Exercise 7.3.3, using backquotes.

7.4.5 Redefine `my-unless`, which was defined in Exercise 7.3.4, using backquotes.

7.4.6 Redefine `when`, which was the answer to Exercise 7.3.5, using backquotes.

7.4.7 Redefine `my-and`, which was defined in Exercise 7.3.6, using backquotes.

7.4.8 Generalized variables (1)
Define a macro := which is similar to `setf`. Like `setf`, the syntax of := is

(:= ⟪location⟫ ⟪expression$_1$⟫)

but only the following types of things are allowed to appear as a ⟪location⟫:

⟪variable⟫
(car ⟪expression$_2$⟫)
(cdr ⟪expression$_2$⟫)

For each type of expression, := should expand as follows:

(setq ⟪variable⟫ ⟪expression$_1$⟫)
(rplaca ⟪expression$_2$⟫ ⟪expression$_1$⟫)
(rplacd ⟪expression$_2$⟫ ⟪expression$_1$⟫)

For example,

```
>(macroexpand '(:= x (foo)))
(SETQ X (FOO))
T

>(macroexpand '(:= (car x) (foo)))
(RPLACA X (FOO))
T

>(macroexpand '(:= (cdr x) (foo)))
(RPLACD X (FOO))
T
```

Answers to Exercises

7.1.1 Since the arguments to `macroexpand` in expressions a through e are not macro forms, the argument itself is the first value and the second value is `NIL`. The first value of f in KCL is

```
(RETURN-FROM NIL 2)
```

(This value could be different on different Common Lisp systems.) The second value would be `T`. Since the argument to `macroexpand` in g is the same as in f, the values will be the same as those in f.

7.1.2

```
(defun my-macroexpand (form)
  (if (and (consp form)
           (symbolp (car form))
           (not (null (macro-function (car form)))))
      (values (my-macroexpand
                (funcall (macro-function (car form))
                         form nil))
              t)
      (values form nil)))
```

7.1.3 It will be clear when the following expression is evaluated:

```
(let ((x nil))
  (do-external-symbols (s (find-package 'lisp) x)
    (when (and (special-form-p s)
               (macro-function s))
      (push s x))))
```

7.2.1 The value of the variable `y` will be the same. The value of the variable `x` will become the following:

a. `(A 2 C D E)`
b. `(A 2 C D E)`
c. `(A B 3 D E)`
d. `(A 2 3 4 5)`
e. `(A B C D 5)`

7.2.2

a. `(rplaca (cdr x) (cadr y))`
b. `(rplaca (cdr x) (cadr y))`
c. `(rplaca (cddr x) (caddr y))`
d. `(rplacd x (cdr y))`
e. `(rplacd (cdddr x) (cddddr y))`

7.2.3

a. 6 b. 7 c. 3 d. 10
e. 4 f. 3 g. 7 h. 0

7.2.4

a. `(1 102 3 4 5)`
b. `(0 2 3 4 5)`
c. `(0 1 2 3 4 5)`
d. `(2 3 4 5)`
e. `(20 10 1 2 3 4 5)`
f. `(4 5)`

7.2.5

a. `((1 . BROTHER) (10 20) (100))`
b. `((1 2 3) (10 21) (100))`
c. `((10 20) (100))`
d. `((1 2 3) (10 20) NIL)`
e. `((1 2 3) (30 10 20) 100))`
f. `((1 2 3) (10 20) ((A B C) 100))`

7.2.6

```
(rplaca (cdr bst) (list n nil nil))

(rplaca (cddr bst) (list n nil nil))
```

should be replaced by

```
(setf (cadr bst) (list n nil nil))

(setf (caddr bst) (list n nil nil))
```

7.3.1

```
>(macroexpand '(num-first x))
[(IF (NUMBERP X) X 0)]
[T]

......

>(foo 1 2 3)
[6]

>(foo 'a 2 3)
[5]

>(foo 'a 'b 'c)
[0]
```

7.3.2

```
(defmacro push-var (form var)
  (list 'setq
        var
        (list 'cons form var)))
```

7.3.3

```
(defmacro my-return (&optional (form nil))
  (list 'return-from nil form))
```

7.3.4

```
(defmacro my-unless (condition &rest body)
  (list 'if
        (list 'not condition)
        (cons 'progn body)))
```

7.3.5

```
(defmacro when (condition . body)
  (cond ((endp body)
         (list 'progn condition nil))
        ((endp (cdr body))
         (list 'if condition (car body)))
        (t (list 'if
                 condition
                 (cons 'progn body)))))
```

7.3.6

```
(defmacro my-and (&rest conditions)
  (labels ((aux (first rest)
             (if (endp rest)
                 first
                 (list 'if
                       first
                       (aux (car rest)
                            (cdr rest))))))
    (if (endp conditions)
        t
        (aux (car conditions)
             (cdr conditions)))))
```

7.4.1

a. `(LIST 'IF CONDITION (CAR BODY))`
b. `(LIST 'SETQ VAR (LIST 'CONS FORM VAR))`
c. `(LIST* 'IF (LIST 'NOT CONDITION) BODY)`

7.4.2

```
(defmacro num-filter (var)
  `(if (numberp ,var) ,var 0))
```

7.4.3

```
(defmacro push-var (form var)
  `(setq ,var (cons ,form ,var)))
```

7.4.4

```
(defmacro my-return (&optional (form nil))
  `(return-from nil ,form))
```

7.4.5

```
(defmacro my-unless (condition &rest body)
  `(if (not ,condition) (progn ,@body)))
```

7.4.6

```
(defmacro when (condition . body)
  (cond ((endp body)
         `(progn ,condition nil))
        ((endp (cdr body))
         `(if ,condition ,(car body)))
        (t `(if ,condition (progn ,@body)))))
```

7.4.7

```
(defmacro my-and (&rest conditions)
  (labels ((aux (first rest)
             (if (endp rest)
                 first
                 `(if ,first
                      ,(aux (car rest)
                            (cdr rest))))))
    (if (endp conditions)
        t
        (aux (car conditions)
             (cdr conditions)))))
```

7.4.8

```
(defmacro := (place value)
  (cond ((symbolp place)
         `(setq ,place ,value))
        ((consp place)
         (case (car place)
           (car `(rplaca ,(cadr place) ,value))
           (cdr `(rplacd ,(cadr place) ,value))))))
```

8 List Processing

Mapping functions are used to call a function repeatedly on a list. There are six mapping functions: **mapcar**, **maplist**, **mapc**, **mapl**, **mapcan**, and **mapcon**.

An alist is a list of pairs and is used to maintain a set of keys together with the value which responds to each key. The function **assoc** is used to operate on alists.

Symbols have property lists. By using property lists to accumulate information, a user can construct a database which has symbols and their properties as keys. The functions **get** or **remprop** are used to operate on the property lists of symbols.

- summary of functions

  (**nth** ⟨nonnegative integers⟩ ⟨list⟩)
  (**nthcdr** ⟨nonnegative integers⟩ ⟨list⟩)

  (**first** ⟨list⟩)
  (**second** ⟨list⟩)
  (**third** ⟨list⟩)
  (**fourth** ⟨list⟩)
  (**fifth** ⟨list⟩)
  (**sixth** ⟨list⟩)
  (**seventh** ⟨list⟩)
  (**eighth** ⟨list⟩)
  (**ninth** ⟨list⟩)
  (**tenth** ⟨list⟩)

```
(rest ⟨list⟩)
(last ⟨list⟩)

(make-list ⟨length⟩ {:initial-element})

(reverse ⟨list⟩)
(nreverse ⟨list⟩)

(member ⟨data⟩ ⟨list⟩ {:test :test-not :key})
(member-if ⟨predicate⟩ ⟨list⟩ {:key})
(member-if-not ⟨predicate⟩ ⟨list⟩ {:key})

(remove ⟨data⟩ ⟨list⟩
  {:test :test-not :key :count :from-end})
(remove-if ⟨predicate⟩ ⟨list⟩
  {:key :count :from-end})
(remove-if-not ⟨predicate⟩ ⟨list⟩
  {:key :count :from-end})

(delete ⟨data⟩ ⟨list⟩
   {:test:test-not :key :count :from-end})
(delete-if ⟨predicate⟩ ⟨list⟩
   {:key :count :from-end})
(delete-if-not ⟨predicate⟩ ⟨list⟩
   {:key :count :from-end})

(mapcar ⟨function⟩ ⟨list⟩ ⟨list⟩...⟨list⟩)
(maplist ⟨function⟩ ⟨list⟩ ⟨list⟩...⟨list⟩)
(mapc ⟨function⟩ ⟨list⟩ ⟨list⟩...⟨list⟩)
(mapl ⟨function⟩ ⟨list⟩ ⟨list⟩...⟨list⟩)
(mapcan ⟨function⟩ ⟨list⟩ ⟨list⟩...⟨list⟩)
(mapcon ⟨function⟩ ⟨list⟩ ⟨list⟩...⟨list⟩)

(assoc ⟨key⟩ ⟨alist⟩ {:key})
(acons ⟨key⟩ ⟨data⟩ ⟨alist⟩)
(pairlis ⟨key list⟩ ⟨data list⟩[⟨alist⟩])

(symbol-plist ⟨symbol⟩)
(get ⟨symbol⟩ ⟨property⟩ [⟨default value⟩])
(remprop ⟨symbol⟩ ⟨property⟩)
(getf ⟨property list⟩ ⟨property⟩ [⟨default value⟩])
(remf ⟨property list⟩ ⟨property⟩)
(get-properties ⟨property list⟩ ⟨list of property⟩)
```

8-1 List Processing Functions

8.1.1 What are the values of the following expressions when the value of `x` is the list `(a b c d e)`:

a. `(third x)`
b. `(sixth x)`
c. `(rest x)`
d. `(nth 3 x)`
e. `(nth 6 x)`
f. `(nthcdr 3 x)`
g. `(nthcdr 6 x)`
h. `(last x)`

8.1.2 Define four functions which are equivalent to the following system functions. Append "`my-`" to the front of the system function.

```
nth
nthcdr
last
make-list
```

You should not worry about cases in which arguments that produce errors in the system functions (for example, a nonnegative integer as the first argument to `nth`) are given to your functions. Self-evident definitions such as

```
(defun my-nth (n list) (nth n list))
```

will not be counted as answers.

8.1.3 Define, using `member`, a predicate `lambda-list-keyword-p` which returns `t` if an object is a lambda list keyword and returns `nil` otherwise. For example,

```
>(lambda-list-keyword-p '&optional)
T

>(lambda-list-keyword-p 'foo)
NIL
```

```
>(lambda-list-keyword-p '(1 2 3))
NIL
```

8.1.4 Define a function, using `member`, called `month` which returns the name of the month when it receives an integer between 1 and 12.

```
>(month 8)
AUG
```

8.1.5 What does the following predicate decide?

```
(defun what? (x)
  (if (member 3 x :test #'>) t nil))
```

8.1.6 Redefine the function `what?` using the function `member-if` that was defined in Exercise 8.1.5.

8-2 Mapping Functions

8.2.1 What are the values of the following expressions:

a. `(mapcar #'1+ '(1 2 3 4 5))`
b. `(mapcar #'list '(1 2 3 4 5))`
c. `(mapcar #'list '(1 2 3 4 5) '(a b c))`
d. `(mapcar #'special-form-p '(cons if car))`
e. `(mapcar #'eval '(1 (+ 1 2) (* 2 3)))`
f.
```
(mapcar #'(lambda (x) (* x 2))
        '(1 2 3 4))
```
g.
```
(let ((x 0))
  (mapcar #'(lambda (y) (incf x))
          '(7 2 3 8)))
```

8.2.2 What are the values of the following expressions:

a. `(maplist #'identity '(one two three))`
b. `(maplist #'reverse '(one two three))`
c.
```
(maplist #'(lambda (x)
             (if (member 'b x) t nil))
         '(a b c d))
```

8.2.3 What are the values of the following expressions:

a.
```
(block foo
  (mapc #'(lambda (x)
            (when (numberp x)
              (return-from foo x)))
        '(a b 3 c 5)))
```

b.
```
(let ((x nil))
  (mapc #'(lambda (y) (push y x))
        '(a b c d))
  x)
```

c.
```
(let ((x '((a . 1) (b . 2) (c . 3))))
  (mapc #'(lambda (y)
            (when (eq (car y) 'b)
              (setf (cdr y) 0)))
        x)
  x)
```

d.
```
(let ((x '(+ 1 2 3 4)))
  (mapl #'(lambda (y) (incf (car y)))
        (cdr x))
  x)
```

8.2.4 What are the values of the following expressions:

a.
```
(mapcan
  #'(lambda (n)
      (make-list n :initial-element n))
  '(4 1 0 2))
```

b.
```
(mapcan
  #'(lambda (n x)
      (make-list n :initial-element x))
  '(4 1 0 2)
  '(a b c d e f g))
```

c.
```
(mapcan
  #'(lambda (x)
      (if (numberp x) (list x) nil))
  '(a b 3 d 5))
```

8.2.5 Processing algebraic expressions

In this example, we will write a program for processing algebraic expressions. Algebraic expression processing is different from numerical computation. Numerical computation is used to find an approximate numerical value by operating numbers directly, such as finding the value of the integral of a function between [0,1]. Algebraic expression processing, on the

other hand, finds a new algebraic expression by transforming the original algebraic expression. In this way we could try to compute the indefinite integral of a function. Since no errors are introduced during the transformation process, a very accurate value for the definite integral can be computed by first symbolically processing the algebraic expression and then producing the accurate indefinite integral.

Suppose we use the following data structures to represent numerical expressions:

1. Symbols for expressing variables
2. Numbers for expressing constant variables
3. Addition
   (+ ⟨⟨numerical expression$_1$⟩⟩ . . . ⟨⟨numerical expression$_n$⟩⟩)
4. Multiplication
   (* ⟨⟨numerical expression$_1$⟩⟩ . . . ⟨⟨numerical expression$_n$⟩⟩)
5. nth power (^ ⟨⟨numerical expression⟩⟩⟨⟨integer⟩⟩)
6. Unary minus (- ⟨⟨numerical expression⟩⟩)

For example,

$$\frac{x^3 - 5}{x + 1}$$

is equivalent to

$$(x^3 + (-5)) \times (x + 1)^{-1}$$

This can be written as

```
(* (+ (^ x 3) -5)
   (^ (+ x 1) -1))
```

Define the function `diff-fmla` which takes a numerical expression f represented as described above and a variable x as an argument and returns the expression in which f is differentiated by x.

```
>(diff-fmla '(* (+ (^ x 3) -5)
                (^ (+ x 1) -1))
            'x)
(+ (* (+ (* 3 (^ X 2) 1) 0)
      (^ (+ X 1) -1))
   (* (+ (^ X 3) -5)
      (* -1 (^ (+ X 1) -2) (+ 1 0))))
```

For your reference, the following formulas are necessary for the differentiation:

Table 8.1 Formulas for differentiation

$$\frac{d}{dx}\langle\langle\text{numerical value}\rangle\rangle = 0$$

$$\frac{d}{dx}\langle\langle\text{variables other than } x\rangle\rangle = 0$$

$$\frac{d}{dx}x = 1$$

$$\frac{d}{dx}(f_1 + f_2 + \cdots f_n) = \frac{d}{dx}f_1 + \frac{d}{dx}f_2 + \cdots + \frac{d}{dx}f_n$$

$$\frac{d}{dx}(f_1 \times f_2 \times \cdots f_n) = \left(\frac{d}{dx}f_1\right) \times f_2 \times \cdots \times f_n$$
$$\times f_1 \times \left(\frac{d}{dx}f_2\right) \times \cdots \times f_n$$
$$\cdots$$
$$\times f_1 \times f_2 \times \cdots \times \left(\frac{d}{dx}f_n\right)$$

$$\frac{d}{dx}(f^n) = n \times f^{n-1} \times \frac{d}{dx}f$$

The simplification of numerical expressions plays a big role in actual numerical expression processing systems. Simplification makes the transformation of numerical expression easier. By outputting the simplified expressions as intermediate results, the results are easier for human beings to understand. Define a function `simplify-fmla` which, when given a numerical expression, simplifies it. You may find the word "simplification" too vague, but please define the function based on what you consider a "simplification." Please include at least the following simplification:

```
>(simplify-fmla
   (diff-fmla '(* (+ (^ x 3) -5)
                  (^ (+ x 1) -1))
              'x))
(+ (* 3 (^ X 2) (^ (+ 1 X) -1))
   (* -1 (+ -5 (^ X 3)) (^ (+ 1 X) -2)))
```

8.2.6 Mini-max

Let's think about a game program which plays against a human being (it could be another program prepared for the same game, but here let's imagine it playing against a human being). Except for very simple games such as the stone picking game in Exercise 5.3.6, it is impossible to make a complete game tree because of time and/or space limitations. In this case it is necessary for a game program to create an incomplete game tree by computing only a few moves ahead. For each possible next move, the program figures out the winning percentages and chooses the next move based on which has the highest chance of winning.

A game program uses a function called an **evaluation function** for each game position in order to calculate its winning percentage. This function estimates the probability of winning for the player in that situation. Suppose we have an evaluation function $f(x)$. Let's call the game program A and its opponent B. The winning percentage $p(x)$ for A in the situation x would be

$$p(x) = f(x)$$

if x is A's turn. And

$$p(x) = 1 - f(x)$$

if x is B's turn. A game program first figures out the winning percentage at the very end of an incomplete game tree using an evaluation function as described above. Based on this percentage, the program figures out the winning percentage for each situation starting with the leaves of the game tree and going up the tree using the process called **mini-max**. Suppose, in a situation x, there are $y_1, \ldots, y_n$ as the next possible situations (see figure 8.1), and the winning percentage for these situations are $p(y_1), \ldots, p(y_n)$. The winning percentage $p(x)$ for the situation x is calculated as follows: Suppose when it is A's turn in the situation x, the greatest value of $p(y_1), \ldots, p(y_n)$ becomes $p(x)$. This is because the program can choose the highest winning percentage since it is A's turn. If it is B's turn in the situation x, B will make the worst move for A, and the smallest value of $p(y_1), \ldots, p(y_n)$ becomes $p(x)$.

Suppose an evaluation function and a (incomplete) game tree are given. Define, using the mini-max process, the functions `estimate-my-turn` and `estimate-his-turn` which figure out the winning percentage for A when the game starts with A's turn and when it starts with B's turn, respectively. A game tree, as in Exercise 5.3.6, is expressed as a list

(⟨⟨situation⟩⟩ ⟨⟨next situation$_1$⟩⟩ ... ⟨⟨next situation$_n$⟩⟩)

If $n \geq 1$, all the possible situations that can follow ⟨⟨situation⟩⟩ should

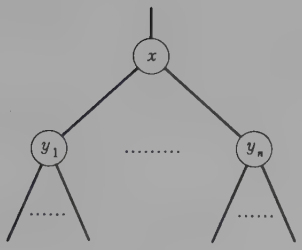


Figure 8.1 mini-max process

be included in ⟨⟨next situation$_1$⟩⟩ ... ⟨⟨next situation$_n$⟩⟩. Otherwise the evaluation function will be used only at

(⟨⟨situation⟩⟩)

In this case, ⟨⟨situation⟩⟩ is the only argument to the evaluation function.

Next, define, using `estimate-my-turn` and `estimate-his-turn`, the functions `estimate-my-turn-for-nim` and `estimate-his-turn-for-nim` when the initial number of stones are given. These figure out the winning percentage of A when A starts the stone-picking game of Exercise 5.3.6 and when B starts the game, respectively.

8-3 Alists

8.3.1 Redefine the function `month`, which was defined in Exercise 8.1.4, using `assoc` rather than `member`.

8.3.2 Interpreter (2)
Enhance `simple-eval`, which was defined in Exercise 6.3.4, so it can evaluate local variable names and call expressions starting with a lambda expression.

⟨⟨SF⟩⟩ : : = ⟨⟨number⟩⟩
| ⟨⟨variable name⟩⟩
| (⟨⟨function name⟩⟩ ⟨⟨SF⟩⟩ . . . ⟨⟨SF⟩⟩)
| (⟨⟨S lambda⟩⟩ ⟨⟨SF⟩⟩ . . . ⟨⟨SF⟩⟩)

⟪S lambda⟫ means a lambda expression which `simple-eval` can evaluate. To keep it simple, the definition should look like

⟪S lambda⟫
::= `(lambda (` ⟪variable⟫ ... ⟪variable⟫ `)` ⟪SF⟫ `)`

In other words, it only allows required parameters, and the body should only have one SF.

Define the function `simple-eval` to take information about the local variable bindings as an optional parameter. This will allow `simple-eval` to refer to local variables. Such information can be expressed as an alist (local variable environment) consisting of a pair containing a variable name and its value.

```
>(simple-eval '(- (* 2 4) 3))
5

>(simple-eval '(- (* 2 4) 3) nil)
5

>(simple-eval
   '(* y y)
   '((x . 1) (y . 2) (z . 3) (y . 0)))
4

>(trace simple-eval)
(SIMPLE-EVAL)

>(simple-eval '((lambda (x) (* x x)) 2))
  1> (SIMPLE-EVAL ((LAMBDA (X) (* X X)) 2))
    2> (SIMPLE-EVAL 2 NIL)
    <2 (SIMPLE-EVAL 2)
    2> (SIMPLE-EVAL (* X X) ((X . 2)))
      3> (SIMPLE-EVAL X ((X . 2)))
      <3 (SIMPLE-EVAL 2)
      3> (SIMPLE-EVAL X ((X . 2)))
      <3 (SIMPLE-EVAL 2)
    <2 (SIMPLE-EVAL 4)
  <1 (SIMPLE-EVAL 4)
4
```

8.3.3 Propositional calculus (1)

Well formed formulas (abbreviated wff(s)) are generated from variables expressing propositions called **propositional variables** and operator symbols called **propositional connectives**. Here, we will define a wff as follows so that a Lisp program can treat it easily:

1. a variable (symbol) is a wff;
2. when x and y are wffs, `(and` x y`)` is a wff;
3. when x and y are wffs, `(or` x y`)` is a wff; and
4. when x is a wff, `(not` x`)` is a wff.

For example,

```
(or p (not p))
```

is a wff according to the above definition.

The truth value of a wff is determined when the truth of the subexpressions of a wff are determined.

1. `(and` x y`)` is true only when x and y are true;
2. `(or` x y`)` is true only when x or y or both of them are true; and
3. `(not` x`)` is true only when x is not true.

Define the function `eval-wff` which takes a wff and an alist that specifies the truth value of the propositional variables contained in that wff and determines the truth value of the wff. True and false are represented as `t` and `nil`, respectively.

```
>(eval-wff 'p
           '((p . t)))
T

>(eval-wff '(or (not p) p)
           '((p . t)))
NIL

>(eval-wff '(and (not p) q)
           '((p . nil) (q . t)))
T
```

```
>(eval-wff '(or (not p) q)
           '((p . t) (q . nil)))
NIL

>(eval-wff '(and p (or q p))
           '((p . nil) (q . nil)))
NIL
```

8.3.4 Propositional calculus (2)

Some wffs of propositional calculus are always true no matter what truth values are assigned to their propositional variables.

```
(or p (not p))
```

is an example. Also if we use

```
(⇒ p q)
```

as an abbreviation of

```
(or (not p) q)
```

then

```
(⇒ (and (⇒ p q) p) q)
```

is always true. This is a well known syllogism.

The easiest method of determining whether a given wff is always true or not is to check whether a wff is true or not for every possible combination of truth values of its propositional variables. This method is not efficient at all, but it can be applied for checking simple wffs. Define the function `validate-wff` for checking whether a wff is always true using this method.

8.3.5 Propositional calculus (3)

Redefine `validate-wff` in Exercise 8.3.4 so that when a given wff is not always true, it returns, as an alist, an example of a combination of propositional variables and truth values for which the wff is not true. If the wff is always true, the function returns `t` as in 8.3.4.

```
>(validate-wff '(or p (not p)))
T
```

```
>(validate-wff '(or (and p q) (not p)))
((P . T) (Q))
```

The latter example shows that the wff becomes false when p is true and q is false.

8-4 Property Lists

8.4.1 Generalized variables (2)

The macro := defined in Exercise 7.4.8 uses only variables, car expressions, and cdr expressions for ⟪location⟫. Change this macro to be a **data-driven** macro so that it can be extended. In other words, besides variables, suppose this macro can use lists starting with some set of symbols

(⟪symbol⟫ ⟪expression_1⟫ ... ⟪expression_n⟫)

as a ⟪location⟫. In such a case, suppose that the function which is used to macro-expand the := expression is the value of the property of ⟪symbol⟫ called expander that has been set up. Two arguments should be given to this function; the first argument is the whole expression for expressing the location, and the second argument is the expression that calculates the new value for that location. The value after the := form is expanded should be the value of the function. For example, suppose the symbol car is given

```
(setf (get 'car 'expander)
      '(lambda (place value)
         `(rplaca ,(cadr place) ,value)))
```

When the system macro-expands

```
(:= (car x) (foo))
```

the first argument (car x) and the second argument (foo) are given to the function defined by the above lambda expression. Its value

```
(rplaca x (foo))
```

is the result of the expansion of the := expression. Redefine := so that it works as described above. Also, set the symbol property for `symbol-value` so that

`(:= (symbol-value` ⟪expression$_1$⟫ `)` ⟪expression$_2$⟫ `)`

expands as

`(set` ⟪expression$_1$⟫ ⟪expression$_2$⟫ `)`

8.4.2 Interpreter (3)
We want to further enhance `simple-eval` which we have already improved in Exercise 8.3.2 so that it can process some special forms.

⟪SF⟫ : := ⟪numbers⟫
| ⟪variable name⟫
| ⟪S special form⟫
| (⟪function name⟫ ⟪SF⟫ . . . ⟪SF⟫)
| (⟪S lambda⟫ ⟪SF⟫ . . . ⟪SF⟫)

⟪S special form⟫ is a list starting with a symbol. However, unlike `eval` in Common Lisp, we do not want to limit the symbols which can be used as special form names, and we want to define a function for evaluating each special form individually.

When `simple-eval` evaluates a list form starting with a symbol, it should check for the property of the symbol called `special-form-handler`. If the symbol does not have such property, `simple-eval` interprets the list form as a function call expression. If the symbol has the property, `simple-eval` interprets the value of the `special-form-handler` as the name of the function to be used when evaluating the special form. It then calls this function and takes the result as the value of the special form. The special form handler is given the special form itself and the variable environment as arguments. For example, if you want to define the special form called `if`, you type

```
>(setf (get 'if 'special-form-handler)
       'if-handler)
IF-HANDLER
```

After this, `simple-eval` recognizes the list form starting with `if` as a special form. The function `if-handler` is used for evaluating the `if` special form. For example, it can be defined as

```
>(defun if-handler (form env)
   (if (simple-eval (cadr form) env)
       (simple-eval (caddr form) env)
       (if (endp (cdddr form))
           nil
           (simple-eval (cadddr form) env))))
IF-HANDLER

>(simple-eval '(if (numberp x) 1 2)
              '((x . 1)))
1

>(simple-eval '(if (consp x) 1 2)
              '((x . 1)))
2

>(simple-eval '(if (consp x) 1)
              '((x . 1)))
NIL
```

Also, you can treat `quote` as a special form if you type

```
>(defun quote-handler (form env)
   (cadr form))
QUOTE-HANDLER

>(setf (get 'quote 'special-form-handler)
       'quote-handler)
QUOTE-HANDLER
```

Define `simple-eval` as described above.

Answers to Exercises

8.1.1

a. C
b. NIL
c. (B C D E)
d. D
e. NIL
f. (D E)
g. NIL
h. (E)

8.1.2 Each function can be defined either recursively or using iteration. Here are the definitions defined recursively:

```
(defun my-nth (n list)
  (if (zerop n)
      (car list)
      (my-nth (1- n) (cdr list))))

(defun my-nthcdr (n list)
  (if (zerop n)
      list
      (my-nthcdr (1- n) (cdr list))))

(defun my-last (list)
  (if (and (consp list)
           (consp (cdr list)))
      (my-last (cdr list))
      list))

(defun my-make-list (n &key
                       (initial-element nil))
  (if (zerop n)
      nil
      (cons initial-element
            (my-make-list (1- n)
                          :initial-element
                          initial-element))))
```

The functions can also be defined using iteration as

```
(defun my-nth (n list)
  (dotimes (i n (car list))
    (pop list)))

(defun my-nthcdr (n list)
  (dotimes (i n list)
    (pop list)))

(defun my-last (list)
  (if (consp list)
      (loop
        (unless (consp (cdr list))
          (return list))
        (pop list))
      list))

(defun my-make-list (n &key
                       (initial-element nil))
  (let ((result nil))
    (dotimes (i n result)
      (push initial-element result))))
```

8.1.3

```
(defun lambda-list-keyword-p (x)
  (if (member x '(&optional &rest &key
                  &allow-other-keys &aux))
      t
      nil))
```

8.1.4

```
(defun month (n)
  (cdar
    (member n
            '((1 . jan) (2 . feb) (3 . mar)
              (4 . apr) (5 . may) (6 . jun)
              (7 . jul) (8 . aug) (9 . sept)
              (10 . oct) (11 . nov) (12 . dec))
            :key #'car)))
```

In order to define this function, `assoc` (see Section 8.3 in *Introduction to Common Lisp*) is usually used rather than `member`. See Exercise 8.3.1.

8.1.5 It takes the list of numbers and determines whether there are any elements smaller than 3.

8.1.6

```
(defun what? (x)
  (if (member-if #'(lambda (y) (< y 3))
                 x)
      t
      nil))
```

8.2.1

a. (2 3 4 5 6)
b. ((1) (2) (3) (4) (5))
c. ((1 A) (2 B) (3 C))
d. (NIL T NIL)
e. (1 3 6)
f. (2 4 6 8)
g. (1 2 3 4)

8.2.2

a. ((ONE TWO THREE) (TWO THREE) (THREE))
b. ((THREE TWO ONE) (THREE TWO) (THREE))
c. (T T NIL NIL)

8.2.3

a. 3
b. (D C B A)
c. ((A . 1) (B . 0) (C . 3))
d. (+ 2 3 4 5)

8.2.4

a. (4 4 4 4 1 2 2)
b. (A A A A B D D)
c. (3 5)

8.2.5

```
(defun diff-fmla (fmla var)
  (cond ((symbolp fmla)
         (if (eq fmla var) 1 0))
        ((numberp fmla) 0)
        (t (case (car fmla)
             (+ (cons '+
                  (mapcar
                    #'(lambda (f) (diff-fmla f var))
                    (cdr fmla))))
             (* (cons '+
                  (mapcar
                    #'(lambda (fmlas) (cons '* fmlas))
                    (diff-*-args (cdr fmla) var))))

             (^ (list '*
                  (caddr fmla)
                  (list '^
                        (cadr fmla)
                        (1- (caddr fmla)))
                  (diff-fmla (cadr fmla) var)))
             (- (list '-
                  (diff-fmla (cadr fmla) var)))))))

(defun diff-*-args (fmlas var)
  (if (null fmlas)
      nil
      (cons (cons (diff-fmla (car fmlas) var)
                  (cdr fmlas))
            (mapcar #'(lambda (fs)
                        (cons (car fmlas) fs))
                    (diff-*-args (cdr fmlas) var)))))
```

Let us explain the differenciation of multiplication expressions. When `diff-fmla` receives the multiplication

(* ⟪numerical expression$_1$⟫ ⟪numerical expression$_2$⟫
... ⟪numerical expression$_n$⟫)

it passes

(⟪numerical expression$_1$⟫ ⟪numerical expression$_2$⟫
... ⟪numerical expression$_n$⟫)

to `diff-*-args`. As a result, `diff-*-args` makes the following list:

```
(( ⟪differential of numerical expression1⟫ ⟪numerical expression2⟫
   ...⟪numerical expressionn⟫ )
 ( ⟪numerical expression1⟫ ⟪differential of numerical expression2⟫
   ...⟪numerical expressionn⟫ )
 ...
 ( ⟪numerical expression1⟫ ⟪numerical expression2⟫
   ...⟪differential of numerical expressionn⟫ ))
```

Arithmetic function symbols are added in the following way:

```
(+ (* ⟪differential of numerical expression1⟫ ⟪numerical expression2⟫
      ...⟪numerical expressionn⟫ )
   (* ⟪numerical expression1⟫ ⟪differential of numerical expression2⟫
      ...⟪numerical expressionn⟫ )
   ...
   (* ⟪numerical expression1⟫ ⟪numerical expression2⟫
      ...⟪differential of numerical expressionn⟫ ))
```

This differentiates the original multiplication expression.

```
(defun simplify-fmla (fmla)
  (if (or (symbolp fmla)
          (numberp fmla))
      fmla
      (case (car fmla)
        (+ (simplify+ (mapcar #'simplify-fmla
                              (cdr fmla))))
        (- (simplify- (simplify-fmla
                        (cadr fmla))))
        (* (simplify* (mapcar #'simplify-fmla
                              (cdr fmla))))
        (^ (simplify^ (simplify-fmla
                        (cadr fmla))
                      (caddr fmla))))))
```

The function `simplify-fmla` takes arithmetic function symbols, simplifies the constituant numerical expressions, and then gives them to the function which simplifies arithmetic function symbol expressions. Let's look at some

easy examples. First the following simplifies an expression for changing the sign of a number:

```
(- ⟪number⟫ ) → -⟪number⟫
(- (- ⟪numerical expression⟫ )) → ⟪numerical expression⟫
```

So you will have

```
(defun simplify- (fmla)
  (cond ((numberp fmla) (- fmla))
        ((and (consp fmla)
              (eq (car fmla) '-))
         (cadr fmla))
        (t (list '- fmla))))
```

The following simplifications are done for multiplication:

```
(^ ⟪numerical expression⟫ 0) → 1
(^ ⟪numerical expression⟫ 1) → ⟪numerical expression ⟫
(^ ⟪number⟫ n) → ⟪number⟫^n
(^ (^ ⟪numerical expression⟫ i) j)
     → (^ ⟪numerical expression⟫ i × j)
```

This becomes

```
(defun simplify^ (fmla n)
  (cond ((zerop n) 1)
        ((= n 1) fmla)
        ((numberp fmla) (expt fmla n))
        ((and (consp (car fmla)
              (eq (car fmla) '^))
         (list '^
               (cadr fmla)
               (* n (caddr fmla))))
        (t (list '^ fmla n))))
```

For addition, the following simplification is done:

```
(+ ⟪number⟫ ) → ⟪number⟫
(+ 0 ⟪numerical expression⟫ ) → ⟪numerical expression⟫
(+ 0 ⟪numerical expression_1⟫ ... ⟪numerical expression_n⟫ )
   → (+ ⟪numerical expression_1⟫ ... ⟪numerical expression_n⟫ )
```

Therefore,

```
(defun simplify+ (fmlas)
  (multiple-value-bind (fmlas1 n)
                       (cc+ (flat+ fmlas))
    (cond ((null fmlas1) n)
          ((zerop n)
           (if (null (cdr fmlas1))
               (car fmlas1)
               (cons '+ fmlas1)))
          (t (list* '+ n fmlas1)))))
```

`flat+` and `cc+` are pre-processing functions. `flat+` changes two nested additions into one addition.

```
(+ ...
   ⟪numerical expression1,i⟫
   (+ ⟪numerical expression2,1⟫ ... ⟪numerical expression2,n⟫ )
   ⟪numerical expression1,i+1⟫
   ...)
→
(+ ...
   ⟪numerical expression1,i⟫
   ⟪numerical expression2,1⟫
   ...
   ⟪numerical expression2,n⟫
   ⟪numerical expression1,i+1⟫
   ...)
```

```
(defun flat+ (fmlas)
  (mapcan #'(lambda (fmla)
              (if (and (consp fmla)
                       (eq (car fmla) '+))
                  (copy-list (cdr fmla))
                  (list fmla)))
          fmlas))
```

`cc+` separates numbers from anything else in the sum. This function returns two values; the expression with numbers removed and the sum of all the numbers.

```
(defun cc+ (fmlas)
  (let ((n 0))
    (values (mapcan #'(lambda (fmla)
                        (cond ((numberp fmla)
                               (incf n fmla)
                               nil)
                              (t (list fmla))))
                    fmlas)
            n)))
```

It is possible to simplify a multiplication just like an addition. Please analyze the simplification of multiplication referring to the simplifications for addition.

```
(defun simplify* (fmlas)
  (multiple-value-bind (fmlas1 n)
                       (cc* (flat* fmlas))
    (cond ((zerop n) 0)
          ((null fmlas1) n)
          ((= n 1)
           (if (null (cdr fmlas1))
               (car fmlas1)
               (cons '* fmlas1)))
          (t (list* '* n fmlas1)))))

(defun flat* (fmlas)
  (mapcan #'(lambda (fmla)
              (if (and (consp fmla)
                       (eq (car fmla) '*))
                  (copy-list (cdr fmla))
                  (list fmla)))
          fmlas))

(defun cc* (fmlas)
  (let ((n 1))
    (values (mapcan
              #'(lambda (fmla)
                  (cond ((numberp fmla)
                         (setq n (* n fmla))
                         nil)
                        (t (list fmla))))
              fmlas)
            n)))
```

8.2.6

```
(defun estimate-my-turn (f game-tree)
  (if (null (cdr game-tree))
      (funcall f (car game-tree))
      (apply #'max
             (mapcar
               #'(lambda (gt)
                   (estimate-his-turn f gt))
               (cdr game-tree)))))

(defun estimate-his-turn (f game-tree)
  (if (null (cdr game-tree))
      (- 1 (funcall f (car game-tree)))
      (apply #'min
             (mapcar
               #'(lambda (gt)
                   (estimate-my-turn f gt))
               (cdr game-tree)))))
```

Since it is easy to build the complete game tree for the stone-picking game, let's give a game tree to the above function. The leaves of the game tree always look like

```
(0)
```

For this case the evaluation function should always return `1` since the winner is the person whose turn it is when all the stones are gone.

```
(defun estimate-my-turn-for-nim (n)
  (estimate-my-turn #'(lambda (x) 1)
                    (nim n)))

(defun estimate-his-turn-for-nim (n)
  (estimate-his-turn #'(lambda (x) 1)
                     (nim n)))
```

If you run `estimate-my-turn-for-nim` with many different arguments, you will find out that when the argument n is of the form $4k+1$ (i.e., when you divide n by 4, you get 1 as the remainder), the function returns `0` and it returns `1` in other cases.

```
>(estimate-my-turn-for-nim 1)
0

>(estimate-my-turn-for-nim 2)
1

>(estimate-my-turn-for-nim 3)
1

>(estimate-my-turn-for-nim 4)
1

>(estimate-my-turn-for-nim 5)
0

>(estimate-my-turn-for-nim 6)
1
```

This shows that when the total number of stones is $4k+1$, the person who moves first should lose and when the total number of stones is other than $4k+1$, the person who moves first should win. It actually happens that way. Please figure out why this is so.

8.3.1

```
(defun month (n)
  (cdr (assoc n
              '((1 . jan)
                (2 . feb)
                (3 . mar)
                (4 . apr)
                (5 . may)
                (6 . jun)
                (7 . jul)
                (8 . aug)
                (9 . sept)
                (10 . oct)
                (11 . nov)
                (12 . dec)))))
```

8.3.2

```
(defun simple-eval (form &optional (env nil))
  (cond ((numberp form) form)
        ((symbolp form) (cdr (assoc form env)))
        ((consp form)
         (cond ((symbolp (car form))
                (apply (car form)
                       (simple-evlis (cdr form) env)))
               ((consp (car form))
                (simple-eval
                  (caddar form)
                  (pairlis (cadar form)
                           (simple-evlis (cdr form) env)
                           env)))))))

(defun simple-evlis (forms env)
  (if (null forms)
      nil
      (cons (simple-eval (car forms) env)
            (simple-evlis (cdr forms) env))))
```

8.3.3

```
(defun eval-wff (wff env)
  (if (symbolp wff)
      (cdr (assoc wff env))
      (case (car wff)
        (and (and (eval-wff (cadr wff) env)
                  (eval-wff (caddr wff) env)))
        (or (or (eval-wff (cadr wff) env)
                (eval-wff (caddr wff) env)))
        (not (not (eval-wff (cadr wff) env)))
        )))
```

8.3.4

```
(defun validate-wff (wff)
  (validate-wff1
    wff
    (get-pvars wff nil)
    nil))
```

```
(defun get-pvars (wff vars)
  (if (symbolp wff)
      (if (member wff vars)
          vars
          (cons wff vars))
      (case (car wff)
        ((and or) (get-pvars (cadr wff)
                             (get-pvars (caddr wff) vars)))
        (not (get-pvars (cadr wff) vars)))))

(defun validate-wff1 (wff pvars bindings)
  (if (endp pvars)
      (eval-wff wff bindings)
      (and (validate-wff1
            wff
            (cdr pvars)
            (cons (cons (car pvars) t)
                  bindings))
           (validate-wff1
            wff
            (cdr pvars)
            (cons (cons (car pvars) nil)
                  bindings)))))
```

Two auxiliary functions are defined here. `get-pvars` takes a wff and returns its second argument together with all the propositional variables in the wff.

```
>(get-pvars '(or p (and p q)) nil)
(P Q)

>(get-pvars '(or p (and p q)) '(r s))
(P Q R S)
```

The function `validate-wff` first makes a list of propositional variables using `get-pvars` and gives this list to another auxiliary function `validate-wff1`. `validate-wff1` takes a wff, a list of propositional variables, and the truth values of the variables in the wff (represented as an alist) which are not part of the propositional variable list. If the propositional variable list is empty, this function evaluates the wff using the given alist. Here we are supposing the truth values for all the propositional variables in the wff appears in the alist. If the propositional variable list is not empty, it calls itself recursively—once using the value true for the truth value of the the

first variable in the list and once using false as the value of the first variable in the list. When the wff is true in both cases, then the truth value of the wff is true. Let's look at the result of tracing:

```
>(trace validate-wff1)
(VALIDATE-WFF1)

>(validate-wff1 '(or p (not p)) '(p) nil)
  1> (VALIDATE-WFF1 (OR P (NOT P)) (P) NIL)
    2> (VALIDATE-WFF1 (OR P (NOT P))
                      NIL
                      ((P . T)))
    <2 (VALIDATE-WFF1 T)
    2> (VALIDATE-WFF1 (OR P (NOT P)) NIL ((P)))
    <2 (VALIDATE-WFF1 T)
  <1 (VALIDATE-WFF1 T)
T
```

In the above example, there is only one propositional variable. The function `validate-wff1` checks whether wff is true or not when `p` is true and then checks whether wff is true when `p` is false. (When tracing, `(p . nil)` is displayed as `(P)`.) Since wff is true in both cases, it has determined that wff is true. Next, let's look at an example with two propositional variables:

```
>(validate-wff1
   '(or (and p q) (or (not p) (not q)))
   '(p q)
   nil)
  1> (VALIDATE-WFF1 ... (P Q) NIL)
    2> (VALIDATE-WFF1 ... (Q) ((P . T)))
      3> (VALIDATE-WFF1 ... NIL
            ((Q . T) (P . T)))
      <3 (VALIDATE-WFF1 T)
      3> (VALIDATE-WFF1 ... NIL
            ((Q) (P . T)))
      <3 (VALIDATE-WFF1 T)
    <2 (VALIDATE-WFF1 T)
    2> (VALIDATE-WFF1 ... (Q) ((P)))
      3> (VALIDATE-WFF1 ... NIL
            ((Q . T) (P)))
      <3 (VALIDATE-WFF1 T)
      3> (VALIDATE-WFF1 ... NIL
            ((Q) (P)))
```

```
        <3 (VALIDATE-WFF1 T)
      <2 (VALIDATE-WFF1 T)
    <1 (VALIDATE-WFF1 T)
T
```

In this example, `validate-wff1` evaluates the wff using the following four combinations:

```
((q . t) (p . t))
((q . nil) (p . t))
((q . t) (p . nil))
((q . nil) (p . nil))
```

As was mentioned in the exercise, this method for checking all the combinations of truth values of the propositional variables is not very effective. There are more effective methods such as the algorithm of Wang or the resolution method. Actual programs use these methods.

8.3.5 When the value of a wff is found to be false, the system should finish the execution of `validate-wff` and return the truth values of the the propositional variables of that time. We recommend using `catch` and `throw`. First, `validate-wff` prepares the catcher.

```
(defun validate-wff (wff)
  (catch 'false
    (validate-wff1
      wff
      (get-pvars wff nil)
      nil)))
```

Then, if `eval-wff` is false, it does a `throw`.

```
(defun validate-wff1 (wff pvars bindings)
  (if (endp pvars)
      (or (eval-wff wff bindings)
          (throw 'false bindings))
      (and ...)))
```

The `(and ...)` part works exactly the same as the answer to 8.3.4.

8.4.1

```
(defmacro := (place value)
  (cond ((symbolp place)
         `(setq ,place ,value))
        ((and (consp place)
              (symbolp (car place)))
         (funcall (get (car place) 'expander)
                  place
                  value))))

(setf (get 'symbol-value 'expander)
      '(lambda (p v) `(set ,(cadr p) ,v)))
```

8.4.2

```
(defun simple-eval (form &optional (env nil))
  (cond ((numberp form) form)
        ((symbolp form) (cdr (assoc form env)))
        ((consp form)
         (cond ((symbolp (car form))
                (if (get (car form)
                         'special-form-handler)
                    (funcall
                      (get (car form)
                           'special-form-handler)
                      form
                      env)
                    (apply (car form)
                           (simple-evlis
                             (cdr form)
                             env))))
               ((consp (car form))
                (simple-eval
                  (caddar form)
                  (pairlis (cadar form)
                           (simple-evlis
                             (cdr form)
                             env)
                           env)))))))
```

9 Declarations

The system can bind global variables as well as local variables. The bindings of local variables are called static bindings, and the bindings of global variables are called dynamic bindings. In order to make dynamic bindings happen, you use **progv**, or make a special declaration in a **let** form, or change symbols to special variables using the **defvar** form.

- summary for functions

(progv ⟨variable list⟩ ⟨initial value list⟩
  ⟪⟨expression⟩⟫ . . . ⟪⟨expression⟩⟫ **)**
(boundp ⟨symbol⟩**)**
(makeunbound ⟨symbol⟩**)**

(proclaim ⟨declaration specifier⟩**)**
(declare ⟪⟨declaration specifier⟩⟫ . . . ⟪⟨declaration specifier⟩⟫ **)**
(documentation ⟨symbol⟩ ⟨kinds⟩**)**
(the ⟪⟨data type⟩⟫ ⟪⟨expression⟩⟫ **)**
(defvar ⟪⟨variable⟩⟫ [⟨initial value⟩ [⟨character string⟩]]**)**
(defconstant ⟪⟨variable⟩⟫ ⟨initial value⟩ [⟨character string⟩]**)**
(defparameter ⟪⟨variable⟩⟫ ⟨initial value⟩ [⟨character string⟩]**)**

9-1 Dynamic Binding

9.1.1 Fill in [ ] with appropriate values:

```
>(setq x 1)
1

>(defun foo () x)
FOO

>(let ((x 2)) x)
[ ]

>(progv '(x) '(3) x)
[ ]

>(let ((x 2)) (foo))
[ ]

>(progv '(x) '(3) (foo))
[ ]
```

9-2 Special Variables

9.2.1 We would like to make x into a special variable just before the interaction of the previous exercise 9.1.1.

```
>(proclaim '(special x))
```

Fill in [ ] in Exercise 9.1.1 with their new values.

9.2.2 What is the value of a variable *env-values* after the following interaction:

```
>(proclaim '(special *env* *env-values*))
NIL

>(setq *env-values* nil
       *env* 0)
0
```

```
>(defun foo (*env*)
   (push *env* *env-values*)
   (incf *env* 100)
   (push *env* *env-values*))
FOO

>(progn
   (push *env* *env-values*)
   (incf *env* 1)
   (push *env* *env-values*)
   (let ((*env* *env*))
        (push *env* *env-values*)
        (incf *env* 2)
        (push *env* *env-values*))
   (push *env* *env-values*)
   (foo *env*)
   (push *env* *env-values*)
   nil)
NIL
```

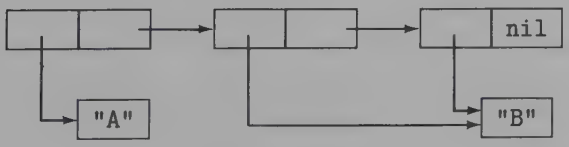
9.2.3 Symbol cells and shallow binding

In order to understand dynamic binding well, it is useful to know how dynamic binding is implemented. There are two methods: **shallow binding** and **deep binding**. Here, we will talk about shallow binding.

First we will explain about the implementation of symbol data. Lists are implemented using cons cells (see Chapter 5 of *Introduction to Common Lisp*). Symbol data is implemented using **symbol cells**. For example, the list

```
(a b b)
```

is implemented as



Symbol cells, unlike cons cells, keep various information for each symbol. Each symbol cell contains the following information:

Package
Print name
Function definition
Variable value
Property list

System functions starting with `symbol-` are used to get such data out of a symbol cell. In order to retrieve the above mentioned five types of data, the following functions are available:

```
symbol-package
symbol-name
symbol-function
symbol-value
symbol-plist
```

Inside of the system, these functions take the pointers to symbol cells and return the objects which are kept in such a cell.

```
(symbol-function '⟨⟨symbol⟩⟩ )
```

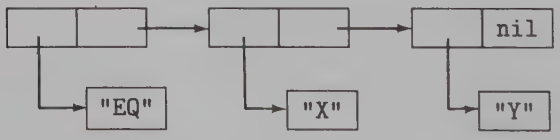
returns the definition of the global function which has ⟨⟨symbol⟩⟩ as a function name.

```
(symbol-value '⟨⟨symbol⟩⟩ )
```

returns the value of the global variable whose name is ⟨⟨symbol⟩⟩. These two operations are frequently used when the system evaluates a form. For example, if you type

```
>(eq x y)
```

the following data structure will be given to the system:



In order to evaluate this form, the system gets the function definition out of the symbol cell `eq` and then gets the variable values out of the symbol cells `x` and `y`.

In the shallow binding method, dynamic binding is implemented by replacing the variable value of the symbol cell. Suppose you type

```
>(let ((*env* 1))
   ... )
```

When the system binds the special variable `*env*`, the system first saves the old variable value inside of the symbol cell of `*env*` and then replaces the variable value with `1`. Therefore, during the execution of the body, unless `*env*` is bound or assigned again, the value of `*env*` stays `1`. At the end of the execution of the body, the system puts the saved old value back in `*env*`. This operation is called **unwinding**. Because of this operation, after the evaluation of the `let` form, the value of `*env*` will not be changed even if an assignment to `*env*` occurs during the execution of the body.

One stack is often used for dynamic binding to remember the variable values. When a dynamic binding occurs, the system pushes the symbol and its old value onto the stack. Then, when leaving that context, the system pops the stack and puts the old value back into the symbol. The reason for keeping the symbol is to remember which variable it had bound. Let's look at the example:

```
>(let ((*env* 2))
   (let ((*env1* 30))
     (let ((*env* 4)))
     ))
```

The outermost `let` does the following:

1.1 puts the symbol `*env*` and its old variable value on the stack and changes the value of `*env*` to 2;
1.2 executes the body;
1.3 does unwind once.

The next `let` does the following:

2.1 puts the symbol `*env1*` and its old variable value on the stack and changes the value of `*env1*` to 30;
2.2 executes the body;
2.3 does unwind once.

The innermost `let` (without a body)

3.1 puts the symbol `*env*` and its old variable value on the stack and changes the value of `*env*` to 4;
3.2 does unwind once.

The above operations are done in the order

$$1.1 \rightarrow 2.1 \rightarrow 3.1 \rightarrow 3.2 \rightarrow 2.3 \rightarrow 1.3.$$

Figure 9.1 shows the state right after 3.1. We are supposing here that the stack was originally empty, and the value of special variables `*env*` and `*env1*` are 0 and 10, respectively. During the three unwinds, the value of `*env*` will go back to 2, the value of `*env1*` will go back to 10, and finally the value of `*env*` will go back to 0. Since unwind needs to be done in the opposite order of binding, the system uses a FILO(First-In, Last-Out) stack.

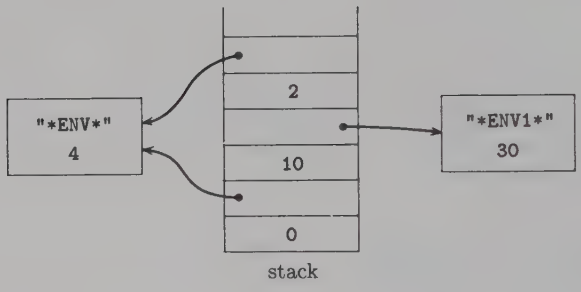


Figure 9.1 Shallow binding using stacks

Show the trace of the execution of the `progn` expression in Exercise 9.2.2 illustrating the use of the stack as in Figure 9.1.

9.2.4 Eight Queens (1)

Let's analyze the "Eight Queens" program which was introduced in Section 9.2 in *Introduction to Common Lisp*. Chess uses an 8×8 board as in Figure 9.2. The queen can proceed forward, backward, left, right, and diagonally, as long as there are no obstacles. Figure 9.3 shows the locations where a queen at (2,3) can go in one turn. The Eight Queens puzzle asks us to place eight queens on the chess board so that each queen does not interfere with any other queen's move. There is more than one answer to this puzzle. One of the answers is shown in Figure 9.4.

Eight Queens was originally posed for a chess board, but it does not have to be a chess board. It can be expanded to the puzzle of n queens on a board of size $n \times n$. The resulting puzzle is called the n Queens puzzle and the function `queen` introduced in *Introduction to Common Lisp* is a program for finding an answer to this puzzle. If you want to solve the Eight Queens puzzle, you just type

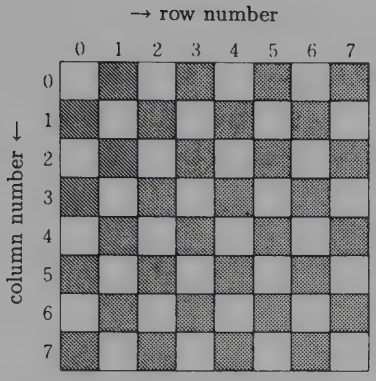


Figure 9.2 Chess board

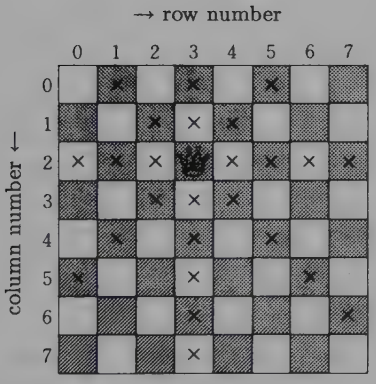


Figure 9.3 movement of Queen

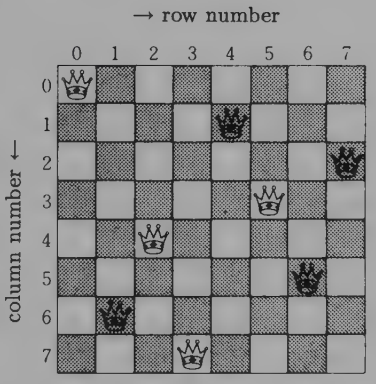


Figure 9.4 a solution to Eight Queens

```
>(queen 8)
(3 1 6 2 5 7 4 0)
```

Each number represents a column number and means that the eight queens should be placed at (0,0), (1,4), (2,7), (3,5), (4,2), (5,6), (6,1), (7,3) from the end to the front of the list. This answer matches the answer shown in Figure 9.4.

The following is the "*n* Queens" program. This is the same program described in *Introduction to Common Lisp*, except that `*n*` rather than `n` is used for the special variable.

```
(proclaim '(special *n*))

(defun queen (*n*)
  (queen1 0 0 nil nil nil))

(defun queen1 (i j column left right)
  (cond ((= i *n*) column)
        ((= j *n*) nil)
        ((or (member j column)
             (member (+ i j) left)
             (member (- i j) right))
         (queen1 i (1+ j)
                 column left right))
        ((queen1 (1+ i) 0
                 (cons j column)
                 (cons (+ i j) left)
                 (cons (- i j) right)))
        ((queen1 i (1+ j)
                 column left right))))
```

queen1 determines where to move the queen from (i, j) when the location of each previously placed queen from line 0 to line $i - 1$ is specified by the parameter `column`. If a possible location is found, it is returned. If one cannot be found, `nil` is returned. `left` is the list of those diagonal lines running from the right top to the left bottom where the next queen cannot be placed. This list contains $x + y$ for each point (x, y) on one of these lines. Similarly, `right` is the list of diagonal lines running from the left top to the right bottom where the next queen cannot be placed. This list contains $x - y$ for each point (x, y) on one of these lines. If there is already some other queen on the diagonal lines specified by `left` or `right`, the next queen cannot go there.

Let's look at a trace of the eight queens computation.

```
>(trace queen1)
(QUEEN1)

>(queen 8)
  1> (QUEEN1 0 0 NIL NIL NIL)
    2> (QUEEN1 1 0 (0) (0) (0))
      3> (QUEEN1 1 1 (0) (0) (0))
        4> (QUEEN1 1 2 (0) (0) (0))
          5> (QUEEN1 2 0 (2 0) (3 0) (-1 0))
            6> (QUEEN1 2 1 (2 0) (3 0) (-1 0))
                  ...
```

The fifth call of `queen1` checks where to place queens below the second line when queens are already placed at (0,0) and (1,2). Since queens exist at (0,0) and (1,2), the value of `left` and `right` are `(3 0)` and `(-1 0)`, respectively.

Based on the above information, analyze the body of `queen1`. Then, write a program for getting all the answers to the n queens problem.

9.2.5 Interpreter (4)

Redefine `simple-eval` defined in Exercise 8.4.2 and the auxiliary function `simple-evlis` so that they do not take a local variable environment as an argument but refer to the special variable `*env*`.

```
>(simple-eval '(- (* 2 4) 3))
5

>(let ((*env* '((x . 2))))
   (simple-eval '(* x x)))
4

>(trace simple-eval simple-evlis)
(SIMPLE-EVAL SIMPLE-EVLIS)

>(let ((*env* '((x . 2) (y . 3))))
   (simple-eval '(* x y)))
  1> (SIMPLE-EVAL (* X Y))
    2> (SIMPLE-EVLIS (X Y))
      3> (SIMPLE-EVAL X)
      <3 (SIMPLE-EVAL 2)
      3> (SIMPLE-EVLIS (Y))
        4> (SIMPLE-EVAL Y)
        <4 (SIMPLE-EVAL 3)
        4> (SIMPLE-EVLIS NIL)
        <4 (SIMPLE-EVLIS NIL)
      <3 (SIMPLE-EVLIS (3))
    <2 (SIMPLE-EVLIS (2 3))
  <1 (SIMPLE-EVAL 6)
6
```

Next, define a macro called `defspecial` which can be used to define a special form that will then be understood by `simple-eval`. For example, if the name of the special form is `quote`, it will be defined as

```
>(defspecial quote form
   (cadr form))
QUOTE-HANDLER
```

and expands to

```
>(macroexpand
   '(defspecial quote form
      (cadr form)))
(PROGN (DEFUN QUOTE-HANDLER (FORM)
         (CADR FORM))
       (SETF (GET 'QUOTE 'SPECIAL-FORM-HANDLER)
             'QUOTE-HANDLER))
```

Finally, use `defspecial` to define the following special forms for use by `simple-eval`:

```
(setq ⟪variable name⟫ ⟪SF⟫ )
(progn ⟪SF⟫...⟪SF⟫ )
(and ⟪SF⟫...⟪SF⟫ )
```

Each of these should have its standard Common Lisp meaning, but `setq` should be restricted to make assignments only to local variables.

9-3 Special Declaration

9.3.1 Most older versions of Lisp languages bound variables dynamically. Therefore, the following happened without using special declarations:

```
% old-fashioned-lisp
OFL (Old Fashioned Lisp)  January 11, 1978

->(defun bar () x)
BAR

->(defun foo (x) (bar))
FOO

->(setq x 1)
1
```

```
->(foo 3)
3
```

When called from `foo`, `bar` referred to the variable (special variable) bound by `foo`. However, when the system compiled the function `foo`, the variable `x` which is bound by `foo` became a statically bound local variable and, as a result, `bar` would no longer refer to that `x`.

```
->(compile 'foo)
FOO

->(foo 3)
Error: The variable X is unbound.
Error signalled by BAR.

Broken at BAR.  Type :H for Help.
->>
```

Lisp was designed like this because compiled code using static binding shoud have better execution speed. On the other hand, when a function was not compiled, in other words, when it was executed using the interpreter, dynamic binding was faster. This choice was necessary when computers were not as powerful as they are at present. In order not to have to simply default to this choice, and in order to request the compiler bind dynamically when appropriate, people have invented the notion of the special declaration.

```
->(defun foo (x)
    (declare (special x))
    (bar))
FOO

->(compile 'foo)
FOO

->(foo 3)
3
```

A local special declaration using `declare` is not necessary in Common Lisp. As a matter of fact, it is better not to use it since it makes a program hard to read. The reason Common Lisp has local special declaration in spite of this fact is to let Common Lisp users continue the heritage developed during the history of Lisp.

Type the following on your system:

```
>(defun bar () x)
BAR

>(defun foo (x)
   (declare (special x))
   (bar))
FOO
```

Then, confirm that the result of

```
>(foo 3)
```

will be the same before and after the compilation of `foo`. Next, change `x` to be a special variable and change the definition of `foo` to

```
>(defun foo (x) (bar))
```

and confirm that the result of

```
>(foo 3)
```

will be the same before and after the compilation of `foo`.

9.3.2 When local variables are never referenced, the compiler will sometimes print a warning. In order to avoid such a warning, you should attach an `ignore` declaration to the form which binds the local variable in question. The declaration specifier for an `ignore` declaration looks like

`(ignore` ⟪variable$_1$⟫ ... ⟪variable$_n$⟫ `)`

Compile the following three functions and see how the compiler reacts on your system:

```
(defun fun1 (x y) (* x x))

(defun fun2 (x y)
      (declare (ignore y))
  (* x x))
```

```
(defun fun3 (x y)
       (declare (ignore y))
  (* x y))
```

9.3.3 Some Common Lisp systems provide documentation strings for special forms, system functions, and macros which can then be used as on-line documentation. Check what kind of documentation strings are available on your system. (See Exercise 4.3.1.)

9.3.4 The type of declaration forms that should be added to variables and expressions in order to improve the execution efficiency depends on your system. `fixnum` declarations are considered to be effective on many Common Lisp systems, especially on general purpose machines. (Refer to Exercise 3.1.7 for `fixnum`.) By adding a declaration such as that, a variable always has a `fixnum` as a value, and if the value of an expression is always `fixnum`, you can expect that a compiled program will get faster.

As an example, let's take the following function which adds all the integers from 1 to n:

```
(defun foo (n)
  (do ((i 1 (1+ i))
       (sum 0 (+ sum i)))
      ((> i n) sum)
    ))
```

Assume that the parameter `n` to this function is small enough so that the sum will never be a `bignum`. Therefore, the three variables `n`, `i`, and `sum` will all have `fixnum` as the type of their values. Thus, we can add the following `fixnum` declarations:

```
(defun foo (n)
    (declare (fixnum n))
  (do ((i 1 (1+ i))
       (sum 0 (+ sum i)))
      ((> i n) sum)
    (declare (fixnum i sum))
    ))
```

Compile these two definitions on your system and compare the execution time.

9-4 Variables and Constants

9.4.1 Do the same calculation which you have done for system functions in Exercise 9.3.3 to the system variables. (See Exercise 4.3.2.)

9.4.2 Type

```
>(defconstant one 1)
```

on your system, evaluate each of the following expressions, and see what happens.

a. `(setq one 2)`
b. `(let ((one 2)) one)`
c. `((lambda (one) one) 2)`

Also, see what happens when you try to compile the functions defined as follows:

d. `(defun foo (one) one)`
e. `(defun bar () (setq one 2))`

Answers to Exercises

9.1.1

```
>(let ((x 2)) x)
[2]

>(progv '(x) '(3) x)
[3]

>(let ((x 2)) (foo))
[1]

>(progv '(x) '(3) (foo))
[3]
```

9.2.1

```
>(let ((x 2)) x)
[2]

>(progv '(x) '(3) x)
[3]

>(let ((x 2)) (foo))
[2]

>(progv '(x) '(3) (foo))
[3]
```

9.2.2

```
(1 101 1 1 3 1 1 0)
```

The values which were pushed are listed in reverse order, 0 from the first push, 1 from the second push, etc.

9.2.3 The state of stacks at the time of each push is shown in Figure 9.5. Since only *env* will be bound, we have omitted the individual entries for keeping symbols on the stack.

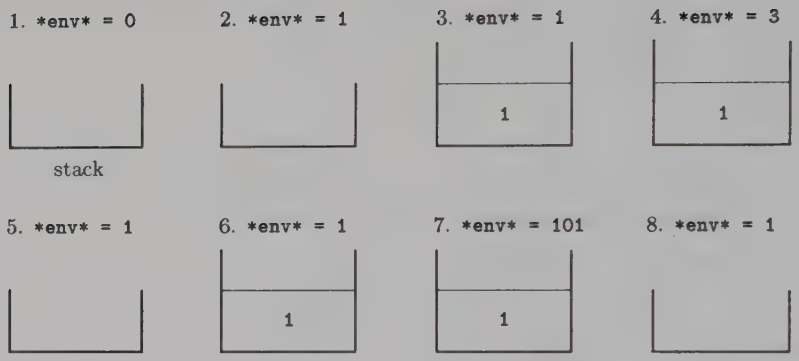


Figure 9.5 Change of stacks

9.2.4 Analysis of the body of queen1 is left to the reader. In order to get all the solutions, queen1 should be rewritten as

```
(defun queen1 (i j column left right)
  (cond ((= i *n*) (print column))
        ((= j *n*))
        ((or (member j column)
             (member (+ i j) left)
             (member (- i j) right))
         (queen1 i (1+ j)
                 column left right))
        (t (queen1 (1+ i) 0
                   (cons j column)
                   (cons (+ i j) left)
                   (cons (- i j) right))
           (queen1 i (1+ j)
                   column left right))))
```

Each time the system has a solution, it prints the solution using print. The above definition considers the possibility of placing another queen at $(i, j+1)$ even when the system gets a solution if a queen is placed at (i, j).

9.2.5

```
>(proclaim '(special *env*))
NIL

>(setq *env* nil)
NIL
```

Then, define as follows:

```
(defun simple-eval (form)
  (cond ((numberp form) form)
        ((symbolp form)
         (cdr (assoc form *env*)))
        ((consp form)
         (cond ((symbolp (car form))
                (if (get (car form)
                         'special-form-handler)
                    (funcall
                      (get
                        (car form)
                        'special-form-handler)
                      form)
                    (apply (car form)
                           (simple-evlis
                             (cdr form)))))
               ((consp (car form))
                (let ((*env*
                       (pairlis
                         (cadar form)
                         (simple-evlis
                           (cdr form)))))
                  (simple-eval (caddar form))
                  ))))))

(defun simple-evlis (forms)
  (if (null forms)
      nil
      (cons (simple-eval (car forms))
            (simple-evlis (cdr forms)))))
```

```
(defmacro defspecial (name parameter . body)
  (let ((handler-name
         (intern (concatenate
                  'string
                  (symbol-name name)
                  "-HANDLER"))))
    `(progn
       (defun ,handler-name (,parameter)
         ,@body)
       (setf (get ',name 'special-form-handler)
             ',handler-name))))

(defspecial setq form
  (let ((val (simple-eval (caddr form))))
    (setf (cdr (assoc (cadr form) *env*)) val)
    val))

(defspecial progn form
  (if (endp (cdr form))
      nil
      (do ((forms (cdr form) (cdr forms)))
          ((endp (cdr forms))
           (simple-eval (car forms)))
        (simple-eval (car forms)))))

(defspecial and form
  (if (endp (cdr form))
      t
      (do ((forms (cdr form) (cdr forms)))
          ((endp (cdr forms))
           (simple-eval (car forms)))
        (unless (simple-eval (car forms))
          (return nil)))))
```

9.3.1

```
>(defun bar () x)
BAR

>(defun foo (x)
   (declare (special x))
   (bar))
FOO
```

```
>(foo 3)
3

>(compile 'foo)
FOO

>(foo 3)
3

>(proclaim '(special x))
NIL

>(defun foo (x) (bar))
FOO

>(foo 3)
3

>(compile 'foo)
FOO

>(foo 3)
3
```

9.3.2 We got the following result using KCL:

```
>(compile 'fun1)
; (DEFUN FUN1 ...) is being compiled.
;; Warning: The variable Y is not used.
FUN1

>(compile 'fun2)
FUN2

>(compile 'fun3)
; (DEFUN FUN3 ...) is being compiled.

;; Warning: The ignored variable Y is used.
```

9.3.3

```
>(let ((x 0))
   (do-external-symbols (s (find-package 'lisp) x)
     (when (documentation s 'function)
       (incf x))))
```

We got the value 646 using KCL.

9.3.4 We got the following result using KCL. The declaration has significantly improved the execution time.

```
>(defun foo (n)
   (do ((i 1 (1+ i))
        (sum 0 (+ sum i)))
       ((> i n) sum)
    ))
FOO

>(compile 'foo)
FOO

>(time (foo 10000))
real time : 2.000 secs
run time  : 1.746 secs
50005000

>(defun foo (n)
    (declare (fixnum n))
   (do ((i 1 (1+ i))
        (sum 0 (+ sum i)))
       ((> i n) sum)
    (declare (fixnum i sum))))
FOO

>(compile 'foo)
FOO

>(time (foo 10000))
real time : 0.000 secs
run time  : 0.027 secs
50005000
```

9.4.1

```
(let ((x 0))
  (do-external-symbols (s (find-package 'lisp) x)
    (when (documentation s 'variable)
      (incf x))))
```

The value of the above expression is 104 on the author's version of KCL.

9.4.2 The author's version of KCL got the following result:

```
>(setq one 2)
Error: Cannot assign to the constant ONE.
Error signalled by SETQ.

Broken at SETQ.  Type :H for Help.
>>:q

Top level.
>(let ((one 2)) one)
Error: ONE is not a variable.
Error signalled by LET.

Broken at LET.  Type :H for Help.
>>:q

Top level.
>((lambda (one) one) 2)
Error: ONE is not a variable.
Error signalled by LAMBDA-CLOSURE.

Broken at LAMBDA-CLOSURE.  Type :H for Help.
>>:q

Top level.
>(defun foo (one) one)
FOO

>(compile 'foo)
; (DEFUN FOO ...) is being compiled.
;;; The constant ONE is being bound.
Failed to compile FOO.
FOO

>(defun bar () (setq one 2))
BAR

>(compile 'bar)
; (DEFUN BAR ...) is being compiled.
;;; The constant ONE is being assigned a value.
Failed to compile BAR.
BAR

>
```

10 Data Types

Numerical data includes integers, ratios, floating-point numbers, and complex numbers.

An array is a row of data. Just like a list, an array can be made using a system function, but it can also be made by directly inputting its elements. An array with only one dimension is called a vector. Vectors and lists have many common operations, and they are both referred to as sequences. A character string is a vector whose elements are of type character.

You can define your own data structures using `defstruct`. Functions for creating data and access functions are defined automatically when `defstruct` is used. You can also specify the field of a structure as a "location" in `setf`.

- summary of functions

  `(integerp` ⟨data⟩`)`
  `(rationalp` ⟨data⟩`)`
  `(floatp` ⟨data⟩`)`
  `(complexp` ⟨data⟩`)`

  `(+` ⟨number⟩ . . . ⟨number⟩`)`
  `(*` ⟨number⟩ . . ⟨number⟩`)`
  `(-` ⟨number⟩ ⟨number⟩ . . . ⟨number⟩`)`
  `(/` ⟨number⟩ ⟨number⟩ . . . ⟨number⟩`)`
  `(1+` ⟨number⟩`)`

(1- ⟨number⟩)
(gcd ⟨integer⟩ ⟨integer⟩...⟨integer⟩)
(lcm ⟨integer⟩ ⟨integer⟩...⟨integer⟩)
(max ⟨number⟩ ⟨number⟩...⟨number⟩)
(min ⟨number⟩ ⟨number⟩...⟨number⟩)
(expt ⟨number⟩ ⟨exponent⟩)
(exp ⟨exponent⟩)
(log ⟨number⟩ [⟨bottom⟩])
(sqrt ⟨number⟩)
(isqrt ⟨nonnegative integer⟩)
(abs ⟨number⟩)

(sin ⟨number⟩)
(cos ⟨number⟩)
(tan ⟨number⟩)
(asin ⟨number⟩)
(acos ⟨number⟩)
(atan ⟨number⟩)

(floor ⟨number⟩ [⟨number⟩])
(ceiling ⟨number⟩ [⟨number⟩])
(truncate ⟨number⟩ [⟨number⟩])
(round ⟨number⟩ [⟨number⟩])
(mod ⟨number⟩ ⟨number⟩)
(rem ⟨number⟩ ⟨number⟩)

(random ⟨positive integer⟩)

(arrayp ⟨data⟩)
(vectorp ⟨data⟩)
(stringp ⟨data⟩)

(make-array ⟨size⟩ {:initial-element :element-type})
(aref ⟨array⟩ ⟨index⟩...⟨index⟩)
(array-rank ⟨array⟩)
(array-dimensions ⟨array⟩)

(copy-seq ⟨sequence⟩)
(concatenate ⟨data structure⟩ ⟨sequence⟩...⟨sequence⟩)
(subseq ⟨sequence⟩ ⟨integer⟩ [⟨integer⟩])

```
(characterp ⟨data⟩)
(alpha-char-p ⟨character⟩)
(upper-case-p ⟨character⟩)
(lower-case-p ⟨character⟩)
(digit-char-p ⟨character⟩)
(alphanumericp ⟨character⟩)

(char= ⟨character⟩ ⟨character⟩...⟨character⟩)
(char/= ⟨character⟩ ⟨character⟩...⟨character⟩)
(char< ⟨character⟩ ⟨character⟩...⟨character⟩)
(char> ⟨character⟩ ⟨character⟩...⟨character⟩)
(char<= ⟨character⟩ ⟨character⟩...⟨character⟩)
(char>= ⟨character⟩ ⟨character⟩...⟨character⟩)

(char-code ⟨character⟩)
(code-char ⟨charactera code⟩)
(char-upcase ⟨character⟩)
(char-downcase ⟨character⟩)

(made-string ⟨length⟩ {:initial-element})
(char ⟨character array⟩ ⟨index⟩)

(string= ⟨character array⟩ ⟨character array⟩)
(string/= ⟨character array⟩ ⟨character array⟩)
(string< ⟨character array⟩ ⟨character array⟩)
(string> ⟨character array⟩ ⟨character array⟩)
(string<= ⟨character array⟩ ⟨character array⟩)
(string>= ⟨character array⟩ ⟨character array⟩)

(string-upcase ⟨string array⟩)
(string-downcase ⟨string array⟩)
(nstring-upcase ⟨string array⟩)
(nstring-downcase ⟨string array⟩)

(defstruct ⟪structure name⟫
  ( ⟪field name⟫ ⟪default expression⟫ )
  ...
  ( ⟪field name⟫ ⟪default expression⟫ ))

(typep ⟨data⟩ ⟨data structure⟩)
```

10-1 Numbers

10.1.1 What is the value of the following expression:

```
(mapcar #'(lambda (x)
            (list (numberp x)
                  (integerp x)
                  (rationalp x)
                  (complexp x)))
        '(foo -12 111111111111111
          3/4 3.14 #c(0 -1)))
```

10.1.2 What is the value of each of the following expressions:

a. `(/ 16 4)`
b. `(/ 16 3)`
c. `(/ 2 1/4)`
d. `(/ -2 1/4)`
e. `(* 2 1/4)`
f. `(* 1/2 1/3)`
g. `(+ 1/2 1/3)`
h. `(+ -1/2 1/4 1/4)`
i. `(- 1/4 1/4)`

10.1.3 What is the value of each of the following expressions:

a. `(list (1+ 20) (1+ 1/2)`
`(1+ 0.5) (1+ #c(0 1)))`
b. `(gcd 12 16)`
c. `(lcm 2 5)`
d. `(max 5/6 0.5 -0.4 -1/3)`
e. `(min 5/6 0.5 -0.4 -1/3)`
f. `(list (expt -2 3) (expt 1/2 3)`
`(expt 0.2 3) (expt #c(0 1) 2))`
g. `(list (sqrt 4) (sqrt 1/4) (sqrt 1.21))`
h. `(list (abs -2) (abs 4) (abs -2/3) (abs #c(3 4)))`

10.1.4 What is the value of each of the following expressions:

a. `(multiple-value-list (floor 11 3))`
b. `(multiple-value-list (floor 11.1))`
c. `(multiple-value-list (floor -11.1))`

d. `(multiple-value-list (floor 1/2))`
e. `(multiple-value-list (floor -1/2))`
f. `(multiple-value-list (ceiling 11 3))`
g. `(multiple-value-list (ceiling 11.1))`
h. `(multiple-value-list (ceiling -11.1))`
i. `(multiple-value-list (ceiling 1/2))`
j. `(multiple-value-list (ceiling -1/2))`
k. `(multiple-value-list (truncate 11.1))`
l. `(multiple-value-list (truncate -11.1))`
m. `(multiple-value-list (truncate 5/2))`
n. `(multiple-value-list (round 2.2))`
o. `(multiple-value-list (round 2.6))`
p. `(multiple-value-list (round -11.1))`
q. `(multiple-value-list (round -11.6))`

10.1.5 Newton method

When an approximate answer a_0 of an equation

$$f(x) = 0$$

is known, you can use Newton's method to find a more accurate answer. This method calculates the array of numbers $\{a_i\}(i = 0, 1, \ldots)$ which is given by

$$a_{i+1} = a_i - \frac{f(a_i)}{f'(a_i)} \quad (i = 0, 1, \ldots)$$

and produces the answer a_n when

$$|a_n - a_{n-1}| < \epsilon$$

for some error $\epsilon(> 0)$. Define the function `poly-solve` which uses Newton's method to solve a polynomial equation in one variable whose coefficients are all numbers

$$c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n = 0$$

Assume the first argument is the polynomial equation expressed in Exercise 6.2.3, the second argument is a_0, and the third argument is the error ϵ. For example, in order to find the approximate value of $2 - x^2 = 0$ you should type

```
>(poly-solve '(2 0 -1) 1.0 0.001)
1.41421356237469
```

or

```
>(poly-solve '(2 0 -1) -1.0 0.001)
-1.41421356237469
```

10.1.6 Using `poly-solve` defined in Exercise 10.1.5, define the function `my-sqrt` which finds the square root of a given positive number. Suppose the error ϵ is 0.001.

10-2 Arrays and Vectors

10.2.1 The function called `month` which was defined in Exercise 8.1.4 and 8.3.1 represents the relation between the number for a month and the name of the month by using an alist. Redefine `month` so that such lists are expressed in vectors.

10.2.2 Fill in [ ] with appropriate values:

```
>(setq magic-square
      #2a((2 9 4) (7 5 3) (6 1 8)))
#2A((2 9 4) (7 5 3) (6 1 8))

>(aref magic-square 2 2)
[ ]

>(aref magic-square 1 1)
[ ]

>(aref magic-square 0 2)
[ ]

>(aref magic-square 2 0)
[ ]

>(array-rank magic-square)
[ ]

>(array-dimensions magic-square)
[ ]

>(setf (aref magic-square 1 1) '?)
?

>magic-square
[ ]
```

10.2.3 Fill in [ ] with appropriate values:

```
>(array-rank #(a b c))
[ ]
```

```
>(array-dimensions #(a b c))
[ ]

>(length #(a b c))
[ ]

>(reverse #(a b c))
[ ]

>(arrayp #(a b c))
[ ]

>(vectorp #(a b c))
[ ]
```

10.2.4 The function `matrix-times`, which was defined in Section 10.2 in *Introduction to Common Lisp*, multiplies only square matrices (matrices where the number of rows and columns are the same) and the number of rows was given to this function as an argument. Redefine `matrix-times` so that it multiplies any two dimensional matrices without specifying the number of rows or columns.

```
>(matrix-times
    #2a((1 1) (1 0))
    #2a((1 2) (3 4)))
#2A((4 6) (1 2))

>(matrix-times
    #2a((1 1) (1 0) (0 1))
    #2a((1 2 3) (4 5 6)))
#2A((5 7 9) (1 2 3) (4 5 6))
```

10.2.5 Suppose there is a vector `*v*` whose elements are all numbers and whose length is `v-size`.

```
>(defconstant v-size 10)
V-SIZE

>(defvar *v* (make-array v-size :initial-element 0))
*V*
```

Define the function `increment-v` which takes a list and increases an element of `*v*` by 1 each time an element of the list is an integer which is a correct index for `*v*`.

```
>(increment-v
   '(one 2 (3 4) 5 -4 1000 2))
NIL

>*v*
#(0 0 2 0 0 1 0 0 0 0)
```

10-3 Characters and Strings

10.3.1 What is the value of the following expression:

```
(mapcar #'(lambda (char)
            (list (alpha-char-p char)
                  (upper-case-p char)
                  (lower-case-p char)
                  (alphanumericp char)))
        '(#\Space #\+ #\3 #\a #\A))
```

10.3.2 `digit-char-p` is a predicate which determines whether a character is "a character expressing a digit." When the predicate is true it returns the number that the character represents.

```
>(digit-char-p #\Page)
NIL

>(digit-char-p #\2)
2
```

"A character expressing a digit" is ambiguous. In the decimal system, there are ten such characters.

```
#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9
```

In base sixteen, in addition to the above, the following are also such characters:

```
#\a #\b #\c #\d #\e #\f
#\A #\B #\C #\D #\E #\F
```

For binary numbers,

```
#\0 #\1
```

are the only such characters. `digit-char-p` takes an optional parameter whose default value is 10 and specifies the radix of the numbers.

```
>(digit-char-p #\3 2)
NIL

>(digit-char-p #\3)
3

>(digit-char-p #\a)
NIL

>(digit-char-p #\a 16)
10
```

Define `my-digit-char-p` which has the same value as `digit-char-p` and which can be given the integers from 2 to 16 as the optional parameter.

10.3.3 Define the function `number-to-chars` which, given a positive integer, returns that number represented as a list of characters using a number base between 2 and 16. The first argument should be the positive integer to convert, and the second argument is an optional parameter which is to be used as the number base and whose default value is 10. For example,

```
>(number-to-chars 123)
(#\1 #\2 #\3)

>(number-to-chars 123 8)
(#\1 #\7 #\3)
```

10.3.4 Define the function `chars-to-number` which does the reverse of `number-to-chars` which was defined in Exercise 10.3.3.

```
>(chars-to-number '(#\1 #\2 #\3))
123

>(chars-to-number '(#\1 #\2 #\3) 8)
83
```

```
>(chars-to-number '(#\1 #\7 #\3) 8)
123
```

10.3.5 Use `char<=` to write expressions which have the same value as the expressions below. Use the ASCII character set.

a. `(alpha-char-p x)`
b. `(upper-case-p x)`
c. `(lower-case-p x)`
d. `(alphanumericp x)`

10.3.6 Find the value of each of the following expressions:

a. `(array-rank "abcde")`
b. `(array-dimensions "abcde")`
c. `(length "Taiichi Yuasa")`
d. `(reverse "Was it a cat I saw")`
e. `(remove #\A "OSAKA")`
f. `(concatenate 'string "to" "get" "her")`
g. `(aref "abcde" 3)`
h. `(char "abcde" 3)`
i. `(subseq "abcdef" 1 3)`

10.3.7 Find the value of the following expression:

```
(mapcar #'(lambda (x)
            (list (arrayp x)
                  (vectorp x)
                  (stringp x)
                  (characterp x)))
        '( #2a((4 3 8) (9 5 1) (2 7 6))
           #(one two three)
           "One Two Three"
           (char "abc" 0)))
```

10.3.8 There is a system function called `sort` for sequences. This function changes the order of elements of a sequence to a specified order and is called in the following manner:

`(sort` ⟨sequence⟩ ⟨predicate⟩`)`

In this function call, ⟨predicate⟩ should be a predicate which takes two arguments and which specifies the order. For example, `#'<` is used to

create sequences in increasing order, and `#'>` is used to create sequences in decreasing order.

```
>(sort '(7 2 3 8) #'<)
(2 3 7 8)

>(sort '(7 2 3 8) #'>)
(8 7 3 2)

>(sort "gazonk" #'char<)
"agknoz"
```

In other words, the sequence which is the value of `sort` is ordered so that

```
(funcall ⟨predicate⟩ ei ei+1)
```

is true for $(i = 0, 1, \ldots, n-1)$ when the elements of the sequence are e_0, $e_1, \ldots, e_n$.

You can also give the keyword parameter called `:key` to `sort`.

```
(sort ⟨sequence⟩ ⟨predicate⟩ :key ⟨key⟩)
```

In this case the function `sort` changes the order so that

```
(funcall ⟨predicate⟩
    (funcall ⟨key⟩ ei)
    (funcall ⟨key⟩ ei+1))
```

becomes true.

```
>(sort '((7 . 3) (2 . 8) (3 . 7) (8 . 2))
       #'< :key #'cdr)
((8 . 2) (7 . 3) (3 . 7) (2 . 8))

>(sort '((7 . 3) (2 . 8) (3 . 7) (8 . 2))
       #'< :key #'car)
((2 . 8) (3 . 7) (7 . 3) (8 . 2))
```

Define, using `sort`, a function `sort-symbols` which, given a list of symbols, returns the symbols in alphabetical order.

```
>(sort-symbols '(you are my sunshine))
(ARE MY SUNSHINE YOU)
```

10-4 Structures

10.4.1 Fill in [ ] with the appropriate values:

```
>(defstruct ship
   name
   (power 100)
   (x 0)
   (y 0))
SHIP

>(setq enterprise
       (make-ship :name "USS Enterprise"))
[ ]

>(setq klingon
       (make-ship :name "Klingon 1" :x 4 :y 4))
[ ]

>(ship-x enterprise)
[ ]

>(ship-power klingon)
[ ]

>(progn
   (incf (ship-x enterprise))
   (decf (ship-power enterprise)))
99

>enterprise
[ ]

>(ship-p enterprise)
[ ]
```

10.4.2 Imagine the structure called `ship` defined in Exercise 10.4.1 is a spaceship which moves on the lattice points of the ordinary x–y plane (i.e., both the x coordinate and the y coordinate are integers). Define the function `move-ship` which moves the spaceship by 1 to one direction.

The first argument should be a `ship` and the second argument should be one of the four symbols `n`, `e`, `s`, and `w` which represent the four directions.

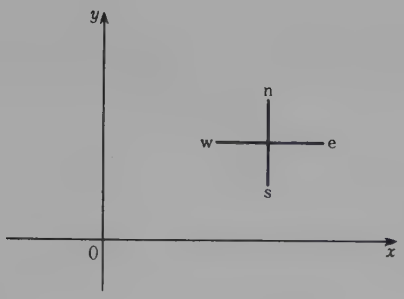


Figure 10.1

Also, arrange it so that the spaceship will decrease its power by 1 when the `move-ship` command is used.

```
>enterprise
#S(SHIP NAME "USS Enterprise" POWER 100 X 0 Y 0)

>(move-ship enterprise 'n)
#S(SHIP NAME "USS Enterprise" POWER 99 X 0 Y 1)

>(move-ship enterprise 'n)
#S(SHIP NAME "USS Enterprise" POWER 98 X 0 Y 2)

>(move-ship enterprise 'e)
#S(SHIP NAME "USS Enterprise" POWER 97 X 1 Y 2)
```

10.4.3 Star Trek

Let's make a game for catching a "Klingon" which has lost control. Suppose you are Captain Kirk who gives Mr. Kato orders on which way to move the Enterprise. You win if you catch the Klingon before the Enterprise loses its power. If your power is gone before catching the Klingon, you lose. Each time Captain Kirk gives a direction, the following thing happens: First, the Enterprise moves in the specified direction. If Captain Kirk catches the Klingon there, he wins. If not, the Klingon moves in a random direction. If, as a result, the Klingon hits the Enterprise, Kirk will also win. If this does not happen and the power of Enterprise is 0, then Kirk loses. Otherwise, the system should display information on the location of the two spaceships at that time.

First, define a predicate `captured?` which checks whether the Enterprise has caught the Klingon or not, in other words, whether the two ships are at the same spot.

Next, define the function **me** which conveys Kirk's orders to Kato. The direction is the only argument. Suppose the Klingon will be controlled automatically by the system function **random** as in

```
(move-ship klingon
  (case (random 5)
    (0 nil)
    (1 'n)
    (2 'e)
    (3 'w)
    (4 's)))
```

Suppose a function called **display** prints information on a spaceship.

```
(defun display ()
  (values enterprise klingon))
```

For example,

```
>(setq enterprise
       (make-ship :name "USS Enterprise"
                  :power 10))
#S(SHIP NAME "USS Enterprise" POWER 10 X 0 Y 0)

>(setq klingon
       (make-ship :name "Klingon 1" :x 2 :y 2))
#S(SHIP NAME "Klingon 1" POWER 100 X 2 Y 2)

>(display)  ; Show where you are
#S(SHIP NAME "USS Enterprise" POWER 10 X 0 Y 0)
#S(SHIP NAME "Klingon 1" POWER 100 X 2 Y 2)

>(me 'n)    ; OK. Go north
#S(SHIP NAME "USS Enterprise" POWER 9 X 0 Y 1)
#S(SHIP NAME "Klingon 1" POWER 99 X 1 Y 2)

            ; The enemy is approaching
>(me 'n)    ; Go North again
#S(SHIP NAME "USS Enterprise" POWER 8 X 0 Y 2)
#S(SHIP NAME "Klingon 1" POWER 98 X 1 Y 2)
            ; The enemy is not moving
>(me 'e)    ; We will win. Go East
WIN         ; We did it.
```

Answers to Exercises

10.1.1

```
((NIL NIL NIL NIL)
 (T T T NIL)
 (T T T NIL)
 (T NIL T NIL)
 (T NIL NIL NIL)
 (T NIL NIL T))
```

10.1.2

a. 4 b. 16/3 c. 8 d. -8 e. 1/2
f. 1/6 g. 5/6 h. 0 i. 0

10.1.3

a. (21 3/2 1.5 #C(1 1)) b. 4 c. 10
d. 5/6 e. -0.4 f. (-8 1/8 0.08 -1)
g. (2.0 0.5 1.1) h. (2 4 2/3 5.0)

10.1.4

a. (3 2) b. (11 0.1) c. (-12 0.9)
d. (0 1/2) e. (-1 1/2) f. (4 -1)
g. (12 -0.9) h. (-11 -0.1) i. (1 -1/2)
j. (0 -1/2) k. (11 0.1) l. (-11 -0.1)
m. (2 1/2) n. (2 0.2) o. (3 -0.4)
p. (-11 -0.1) q. (-12 0.4)

10.1.5

```
(defun poly-solve (poly est error)
  (let ((poly* (poly-diff poly)))
    (do ((prev est x)
         (x (- est (/ (poly-eval poly est)
                      (poly-eval poly* est)))
            (- x (/ (poly-eval poly x)
                    (poly-eval poly* x)))))
        ((< (abs (- x prev)) error)
         x)
      )))
```

The functions `poly-diff` and `poly-eval` are already defined in Exercises 6.2.3 and 6.2.4.

10.1.6

```
(defun my-sqrt (x)
  (poly-solve `(,x 0 -1) 1.0 0.001))
```

10.2.1

```
(defun month (n)
  (aref #(nil jan feb mar apr may jun
              jul aug sept oct nov dec)
        n))
```

The index of a vector starts at 0. Since the month 0 does not exist, you can put any random symbol as the first element of the vector. We used `nil` here.

10.2.2

```
>(aref magic-square 2 2)
[8]

>(aref magic-square 1 1)
[5]

>(aref magic-square 0 2)
[4]

>(aref magic-square 2 0)
[6]

>(array-rank magic-square)
[2]

>(array-dimensions magic-square)
[(3 3)]

>(setf (aref magic-square 1 1) '?)
?

>magic-square
[#2A((2 9 4) (7 ? 3) (6 1 8))]
```

10.2.3

```
>(array-rank #(a b c))
[1]

>(array-dimensions #(a b c))
[(3)]

>(length #(a b c))
[3]

>(reverse #(a b c))
[#(C B A)]

>(arrayp #(a b c))
[T]

>(rectorp #(a b c))
[T]
```

10.2.4

```
(defun matrix-times (a b)
  (let* ((p (car (array-dimensions a)))
         (q (cadr (array-dimensions a)))
         (r (cadr (array-dimensions b)))
         (c (make-array (list p r))))
    (dotimes (i p c)
      (dotimes (j r)
        (let ((s 0))
          (dotimes (k q)
            (incf s (* (aref a i k)
                       (aref b k j))))
          (setf (aref c i j) s))))))
```

10.2.5

```
(defun increment-v (list)
  (dolist (element list)
    (when (and (integerp element)
               (<= 0 element)
               (< element v-size))
      (incf (aref *v* element)))))
```

10.3.1

```
((NIL NIL NIL NIL)
 (NIL NIL NIL NIL)
 (NIL NIL NIL T)
 (T NIL T T)
 (T T NIL T))
```

10.3.2

```
(defun my-digit-char-p (char
                        &optional (radix 10))
  (case char
    (#\0 0)
    (#\1 1)
    (#\2 (if (> radix 2) 2 nil))
    (#\3 (if (> radix 3) 3 nil))
    (#\4 (if (> radix 4) 4 nil))
    (#\5 (if (> radix 5) 5 nil))
    (#\6 (if (> radix 6) 6 nil))
    (#\7 (if (> radix 7) 7 nil))
    (#\8 (if (> radix 8) 8 nil))
    (#\9 (if (> radix 9) 9 nil))
    ((#\a #\A) (if (> radix 10) 10 nil))
    ((#\b #\B) (if (> radix 11) 11 nil))
    ((#\c #\C) (if (> radix 12) 12 nil))
    ((#\d #\D) (if (> radix 13) 13 nil))
    ((#\e #\E) (if (> radix 14) 14 nil))
    ((#\f #\F) (if (> radix 15) 15 nil))
    (otherwise nil)))
```

10.3.3

```
(defun number-to-chars (number
                        &optional (radix 10))
  (do ((n number (floor n radix))
       (result nil))
      ((zerop n) (if (null result) '(#\0) result)
    (push (cdr (assoc (mod n radix)
                      '((0 . #\0) (1 . #\1)
                        (2 . #\2) (3 . #\3)
```

```
                        (4 . #\4) (5 . #\5)
                        (6 . #\6) (7 . #\7)
                        (8 . #\8) (9 . #\9)
                        (10 . #\a) (11 . #\b)
                        (12 . #\c) (13 . #\d)
                        (14 . #\e) (15 . #\f))
                      ))
          result)))
```

10.3.4

```
(defun chars-to-number (chars
                        &optional (radix 10))
  (let ((result 0))
    (dolist (char chars result)
      (setq result (+ (* result radix)
                      (digit-char-p char
                                    radix))))))
```

10.3.5

```
a. (or (char<= #\A x #\Z)
       (char<= #\a x #\z))
b. (char<= #\A x #\Z)
c. (char<= #\a x #\z)
d. (or (char<= #\A x #\Z)
       (char<= #\a x #\z)
       (char<= #0 x #\9))
```

10.3.6

a. 1 b. (5) c. 13
d. "was I tac a ti saW" e. "OSK" f. "together"
g. #\d h. #\d i. "bc"

10.3.7

```
((T NIL NIL NIL)
 (T T NIL NIL)
 (T T T NIL)
 (NIL NIL NIL NIL))
```

Pay attention to the fact that the last `(char "acb" 0)` is a list.

10.3.8

```
(defun sort-symbols (symbols)
  (sort symbols #'string< :key #'symbol-name))
```

10.4.1

```
>(setq enterprise
       (make-ship :name "USS Enterprise"))
[#S(SHIP NAME "USS Enterprise" POWER 100 X 0 Y 0)]

>(setq klingon
       (make-ship :name "Klingon 1" :x 4 :y 4))
[#S(SHIP NAME "Klingon 1" POWER 100 X 4 Y 4)]

>(ship-x enterprise)
[0]

>(ship-power klingon)
[100]

>(progn
   (incf (ship-x enterprise))
   (decf (ship-power enterprise)))
99

>enterprise
[#S(SHIP NAME "USS Enterprise" POWER 99 X 1 Y 0)]

>(ship-p enterprise)
[T]
```

10.4.2

```
(defun move-ship (ship direction)
  (case direction
    (n (incf (ship-y ship)))
    (e (incf (ship-x ship)))
    (w (decf (ship-x ship)))
    (s (decf (ship-y ship))))
  (decf (ship-power ship))
  ship)
```

10.4.3

```
(defun captured? ()
  (and (= (ship-x enterprise)
          (ship-x klingon))
       (= (ship-y enterprise)
          (ship-y klingon))))

(defun me (direction)
  (move-ship enterprise direction)
  (when (captured?) (return-from me 'win))
  (move-ship klingon
             (case (random 5)
               (0 nil)
               (1 'n)
               (2 'e)
               (3 'w)
               (4 's)))
  (cond ((captured?) 'win)
        ((plusp (ship-power enterprise))
         (display))
        (t 'loose)))
```

11 Input and Output

Standard input and output are done to the streams which are the values of the system variables `*standard-input*` and `*standard-output*`. These streams are usually connected to the terminal. You can change where you will read or write by specifying to the I/O functions an input stream from a file or an output stream to a file.

`read` is used to input data. `write`, `print`, etc. are used to output data. `read-char` and `write-char` is used to input and ouput characters. You can specify some parameters for controlling the output in `write`. You can also change the standard output control by changing the value of the system variable corresponding to each parameter.

The `format` function is used to output in some particular format.

- summary of functions

```
(write ⟨data⟩
  {:stream :escape :radix :base :circle
   :pretty :length :level :case :gensym
   :array})

(princ ⟨data⟩ [⟨stream⟩])
(prin1 ⟨data⟩ [⟨stream⟩])
(print ⟨data⟩ [⟨stream⟩])
```

```
(pprint ⟨data⟩ [⟨stream⟩])
(terpri [⟨stream⟩])
(fresh-line [⟨stream⟩])
(finish-output [⟨stream⟩])

(read [⟨stream⟩ [⟨truth value⟩ [⟨final value⟩]]])
(read-preserving-whitespace
  [⟨stream⟩ [⟨truth value⟩ [⟨final value⟩]]])

(write-char ⟨character⟩ [⟨stream⟩])
(read-char [⟨stream⟩])
(unread-char ⟨character⟩ [⟨stream⟩])
(peek-char [⟨peek-type⟩ [⟨stream⟩]])

(open ⟨file name⟩ {:direction})
(close ⟨stream⟩)
(with-open-file ( ⟪⟨variable⟩⟫ ⟨file name⟩
                  ⟨option⟩...⟨option⟩)
  ⟪⟨expression⟩⟫...⟪⟨expression⟩⟫ )
(probe-file ⟨file name⟩)
(delete-file ⟨file name⟩)
(rename-file ⟨file name⟩ ⟨file name⟩)

(with-input-from-string ( ⟪⟨variable⟩⟫ ⟨character string⟩)
  ⟪⟨expression⟩⟫...⟪⟨expression⟩⟫ )
(with-output-to-string ( ⟪⟨variable⟩⟫ )
  ⟪⟨expression⟩⟫...⟪⟨expression⟩⟫ )
(write-to-string ⟨data⟩)

(get-macro-character ⟨character⟩)
(set-macro-character ⟨character⟩ ⟨function⟩)

(format ⟨stream⟩ ⟨format character string⟩
  ⟨argument⟩...⟨argument⟩)
```

- summary of variables

```
*standard-output*
*standard-input*
*terminal-io*

*print-escape*
*print-radix*
*print-base*
```

```
*print-circle*
*print-pretty*
*print-level*
*print-length*
*print-case*
*print-gensym*
*print-array*

*read-base*
*readtable*
```

11-1 Streams

11.1.1 Check the initial value of each of the following variables on your system:

```
*terminal-io*
*standard-output*
*standard-input*
```

11.1.2 A **broadcast stream** is an output stream composed of several output streams. Output to a broadcast stream will be done to all the streams it contains. A broadcast stream can be made by giving the component output streams to the system function `make-broadcast-stream`.

```
(make-broadcast-stream ⟨stream1⟩
                       ...
                       ⟨streamn⟩)
```

If a stream s appears twice as an argument to `make-broadcast-stream`, the output to the resulting broadcast stream will be output to s twice. Check how this duplication works on your system. Change standard output as follows:

```
>(setq *standard-output*
       (make-broadcast-stream
         *standard-output*
         *standard-output*))
```

11-2 Outputting Data

11.2.1 For each of the following examples, show the form of the calls to `write` which will output the expression on the right when it gets the form on the left as an argument.

a. `|Common Lisp|`	`Common Lisp`
b. `#\a`	`a`
c. `(#\a "What?")`	`(a What?)`
d. `10`	`1010`
e. `10`	`#b1010`
f. `10`	`10.`
g. `IF`	`if`
h. `IF`	`If`
i. `(a b c d e f)`	`(A B C ...)`

11.2.2 Show what is displayed when the system evaluates the following expression:

```
(mapc #'(lambda (x)
          (princ x)
          (prin1 x)
          (print x))
  '(|123|
    123
    #\Z
    "Hey, You!"
    (a #\b |c|)
    #(a #\b |c|)))
```

11.2.3 Define the function `write-strings` which takes one or more character strings and outputs them as shown below. Make this function in such a way that it does not return a value.

```
>(write-strings "red")
red.

>(write-strings "red" "green")
red and green.

>(write-strings "red" "green" "blue")
red, green, and blue.

>(write-strings "red" "green" "blue" "white")
red, green, blue, and white.
```

11.2.4 Fill in [ ] with the appropriate data:

```
>*print-base*
10

>*print-radix*
NIL

>(setq *print-base* 2)
[ ]

>128
[ ]

>(setq *print-radix* t)
T

>128
[ ]

>(setq *print-base* 10)
[ ]

>128
[ ]
```

11.2.5 Fill in [ ] with the appropriate data:

```
>*print-length*
NIL

>*print-level*
NIL

>(setq x (make-list 8))
(NIL NIL NIL NIL NIL NIL NIL NIL)

>(setq y (let ((z nil))
           (dotimes (i 4 z) (push z z))))
((((NIL) NIL) (NIL) NIL) ((NIL) NIL) (NIL) NIL)

>(setq *print-length* 3)
3

>x
[ ]
```

```
>y
[ ]

>(setq *print-level* 3)
3

>x
[ ]

>y
[ ]
```

11-3 Inputting Data

11.3.1 Fill in [ ] with the appropriate data:

```
>*read-base*
10

>*print-base*
10

>'(#b10 #o10 #x10 #xf)
[ ]

>(setq *read-base* 2)
2

>(list 10 (+ 10 100) (+ 10. 10) #o10)
[ ]

>(setq *print-base* 2.)
[ ]

>(cons (+ 100 10) (- 100 1))
[ ]
```

11.3.2 Define the function `convert` which reads a number from the terminal using i as the number base and prints the number in base j. The first argument should be i and the second argument j. Both arguments should be integers between 2 and 16. The function `convert` should stop executing when it reads something which is not a number. For example,

```
>(convert 10 2)
CONVERT>8
1000

CONVERT>7
111

CONVERT>end
END

>(convert 16 10)
CONVERT>fff
4095

CONVERT>10
16

CONVERT>(bye bye)
(BYE BYE)

>
```

11-4 Input and Output of Characters

11.4.1 Write a program called `calc` which works just like a calculator. `calc` should read, from the terminal, a sequence of characters of the following form:

⟪numerical string⟫ ⟪operator⟫ ⟪numerical string⟫ ...
⟪operator⟫ ⟪numerical string⟫.

Here ⟪numerical string⟫ is a sequence of digits. There should not be any alphabetic characters between the digits, just the characters 0 through 9. ⟪operator⟫ is any one of the symbols `+`, `-`, `*`, or `/`. The symbol `/` means division using `round` (see Section 10.1 in *Introduction to Common Lisp*). When it reads a ".", `calc` should read ⟪numerical string⟫ and

⟨⟨operator⟩⟩ alternatively, process them from the left, and return the result of the calculation.

```
>(calc)
1+2.
3

>(calc)
10+20*3/30+5.
8
```

This function should ignore characters other than digits in front of ⟨⟨numerical string⟩⟩ and characters other than "+", "-", "*", and "/" instead of "." or ⟨⟨operator⟩⟩.

```
>(calc)
   10 + 20*3
/30      +5
.
8
```

First, define the function `read-number` which reads a ⟨⟨numerical string⟩⟩ and returns the integer expressed by this string. Then, define the function `read-op` which reads an ⟨⟨operator⟩⟩ or "." and returns what it read as a character object. Both functions should not have any arguments. Lastly, define the function `calc` using these functions.

11.4.2 Eight Queens (2)

Define the function `print-queens` which displays a solution to the n Queens problem in a visually simple way. The first argument should be a positive integer n and the second argument should be a solution to the n Queens problem. The solution should be given as in Exercise 9.2.4.

```
>(print-queens 8 '(3 1 6 2 5 7 4 0))
Q-------
----Q---
-------Q
-----Q--
--Q-----
------Q-
-Q------
---Q----
NIL
```

11-5 File System

11.5.1 The `trace` function which appeared several times in this book actually outputs the trace information to a stream called `*trace-output*` (to be more precise, the stream which is the value of the variable `*trace-output*`—see the answer to Exercise 11.1.1). If the value of this variable is changed to an output stream to a file, the result of trace will be output to that file. Define the function `begin-trace-to-file`, which starts tracing to a file, and the function `end-trace-to-file`, which ends such tracing. For example, these functions can be used as follows:

```
>(begin-trace-to-file "trace.out")
#<output stream "trace.out">

>(defun fact (n)
   (if (zerop n)
       1
       (* n (fact (1- n)))))
FACT

>(trace fact)
(FACT)

>(fact 3)
6

>(fact 2)
2

>(end-trace-to-file)
NIL

>(fact 2)
  1> (FACT 2)
    2> (FACT 1)
      3> (FACT 0)
      <3 (FACT 1)
    <2 (FACT 1)
  <1 (FACT 2)
2

>
```

At the end of the above interaction, the file `trace.out` should contain the following:

```
1> (FACT 3)
  2> (FACT 2)
    3> (FACT 1)
      4> (FACT 0)
      <4 (FACT 1)
    <3 (FACT 1)
  <2 (FACT 2)
<1 (FACT 6)
1> (FACT 2)
  2> (FACT 1)
    3> (FACT 0)
    <3 (FACT 1)
  <2 (FACT 1)
<1 (FACT 2)
```

11.5.2 Write to a file all the external symbols of the `lisp` package as a pair consisting of a symbol and its value. Use the file named `symbols.out`.

Each line of the file should look like

⟪symbol name⟫ = ⟪value⟫

if the symbol has a value or

⟪symbol name⟫ : `unbound`

if it does not.

11.5.3 Suppose a file, for example, `numbers.in`, contains several numbers. Write an expression to find the sum of such numbers. For example, if `numbers.in` contains the following numbers:

```
1 2
3 4 5 6
7 8 9
```

then the expression would return

```
>⟪necessary expression⟫
45
```

11.5.4 Define the function `copy-file` which makes a copy of the content of a file. The first argument should be the file name, and the second argument should be the name of a new file to be made. Both files should consist entirely of strings of characters.

11.5.5 Define the function `count-lines` which counts the number of lines in a file. If the file ends with the line separator `#\Newline`, it should return the total number of `#\Newlines` in the file. If the file does not end with a line separator, it should return the total number of `#\Newlines` plus 1. However, if the file is empty, in other words, if it does not contain any letters, the number of lines should be 0.

11.5.6 Define the function `backup-file` which makes a backup file for the specified file. The name of the backup file should be the name of the specified file plus the extension `.BAK`. For example, the backup file for `foo.lsp` should be named `foo.lsp.BAK`. `backup file` should erase an old backup file if one already exists and then copies the file to a new version of the backup file.

The kind of character strings allowed in a file name depends on your operating system. Your system may not allow two "."s in a file name like `foo.lsp.BAK`. In such a case, find another naming convention for the backup file and define `backup-file` using it.

11.5.7 Define the function `show-file` which displays on the screen the nth line of a file and the following $i-1$ lines ($n \geq 1$, $i \geq 1$). For example, suppose the file `foo` has the following content:

```
this is the first line.
this is the second line.
this is the third.
this is next to the third.
this is the last line.
```

This function should work as follows:

```
>(show-file "foo" 2 3)
this is the second line.
this is the third.
this is next to the third.
NIL
```

If a file does not have n lines, this function should display nothing:

```
>(show-file "foo" 6 1)
NIL
```

If a file contains fewer than i lines after the nth, it should display the rest of the file:

```
>(show-file "foo" 4 3)
this is next to the third.
this is the last line.
NIL
```

11-6 Input and Output to Strings

11.6.1 Define the function `append-strings`, using `with-output-to-string`, which takes an arbitrary number of character strings as arguments and makes a new string by concatenating the argument strings.

```
>(append-strings "Who" " " "are" " " "you" "?")
"Who are you?"
```

11.6.2 Define the function `count-words-in-string` which takes one character string and returns the number of words in the string. For this example, "words" should consist of alphabetic characters and digits from 0 to 9. For example,

```
>(count-words-in-string " Who are you ?")
3
```

First, define the function `find-word` which takes an input stream and moves its pointer to the beginning of the next word. `find-word` should return the first character of the next word as a value; however, if it cannot find the next word and reaches the end of the stream, it should return `nil`.

```
>(with-input-from-string (in " What ?")
   (list (find-word in)
         (read-char in)))
(#\W #\W)

>(with-input-from-string (in " ?")
   (list (find-word in)
         (read-char in nil 'eos)))
(NIL EOS)
```

Suppose the pointer now rests at the begining of a word. Define the function `read-word` which skips the letters in this word.

```
>(with-input-from-string (in " What's this ?")
   (list (find-word in)
         (peek-char nil in)
         (read-word in)
         (read-char in)))
(#\W #\W NIL #\')
```

Finally, define the function `count-words` which takes a stream instead of a character string and returns the number of words of the stream.

```
>(with-input-from-string (in " Who are you ?")
   (count-words in))
3
```

Define `count-words-in-string` using the above functions.

11.6.3 Some readers may have wondered what input and output to strings could be used for when they read Section 11.6 of *Introduction to Common Lisp*. By doing Exercise 11.6.2, some may already have found out the answer. The facility to do input and output to strings is very useful for testing the input and output of a program. A program which does input and output to files must use several functions which take streams as arguments. The opportunity to do input and output from strings allows a user to test each function without making test files. In Exercise 11.6.2, we have already tested `count-words`. Define the function `count-words-in-file` which finds the number of words in a file. Its only argument should be a character string which specifies the file name.

11-7 Read Macros

11.7.1 Fix it so that "~" can be used for negation. In other words, rewrite the read table so that when `read` sees

~⟪data⟫

it will return

`(not` ⟪data⟫ `)`

For example,

```
>'~(consp x)
(NOT (CONSP X))

>(when ~(consp 1) 'not-cons)
NOT-CONS
```

11.7.2 `get-macro-character` returns `nil` if the specified character is not a macro character. Check what characters are macro characters on your system.

11-8 Sharp Sign Macro

11.8.1 The system function `set-dispatch-macro-character` is used to define a sharp sign macro. This function is used in the same way as `set-macro-character`.

```
(set-dispatch-macro-character #\#
   ⟨character⟩
   ⟨function⟩)
```

When `read` reads #⟨character⟩, ⟨function⟩ is called with the following argument:

`(funcall` ⟨function⟩ ⟨input stream⟩ ⟨characters⟩ `nil)`

A vector is usually input as

`#(` ⟪element$_1$⟫ ... ⟪element$_n$⟫ `)`

Define a sharp sign macro `#v` so that a vector can also be input as

`#v(` ⟪element$_1$⟫ ... ⟪element$_n$⟫ `)`

For example,

```
>#(a b c)
#(A B C)

>#v(a b c)
#(A B C)
```

11-9 Formatted Output

11.9.1 Define the function `display-ship` which takes a `ship` structure, which was defined in Exercise 10.4.1, as an argument and displays it as follows with one call to `format`.

⟪name⟫ `at` (x, y) `with` ⟨power⟩ `units of power.`

For example,

```
>enterprise
#S(SHIP NAME "USS Enterprise" POWER 97 X 1 Y 2)

>(display-ship enterprise)
USS Enterprise at (1,2) with 97 units of power.
NIL
```

11.9.2 Rewrite the form given in Exercise 11.5.2 using `format`.

11.9.3 Change `backup-file` which was defined in Exercise 11.5.6 so that it saves old backup files. There are many methods to do this. We will use the following one: the name of the backup files will have the extensions `.0`, `.1`, ..., `.10`, `.11`, ... starting from the oldest file. We will not use the old file name plus `.BAK`. The names for `foo.lsp`, for example, will become `foo.lsp.0`, `foo.lsp.1`, and so on. The new function `backup-file` will create a backup file name from the oldest, check if such file name exists or not, and use this name for the next backup file name if it does not exist.

11.9.4 The system function **error** (see A2.2 of *Introduction to Common Lisp*) takes a formatted character string as an argument.

```
>(defun square (x)
   (if (numberp x)
       (* x x)
       (error "~S is not a number." x)))
SQUARE

>(square 'two)
Error: TWO is not a number.
Error signalled by IF.

Broken at ERROR.  Type :H for Help.
>>:q

Top level.
>
```

Improve the function **convert** which was defined in Exercise 11.3.2 so that the system will output an appropriate error message when the arguments are not correct. For example,

```
>(convert 'two 10)
Error: TWO is not an integer.
Error signalled by COND.

Broken at ERROR.  Type :H for Help.
>>:q

Top level.
>(convert 0 10)
Error: 0 is not between 2 and 16.
Error signalled by COND.

Broken at ERROR.  Type :H for Help.
>>:q

Top level.
>(convert 2 -3)
Error: -3 is not between 2 and 16.
Error signalled by COND.
```

```
Broken at ERROR.  Type :H for Help.
>>
```

11.9.5 The system function `warn` works almost the same as `error`, but usually it only beeps without entering the error handler.

```
>(defun square (x)
   (if (numberp x)
       (* x x)
       (warn "~S is not a number." x)))
SQUARE

>(square 'two)
Warning: TWO is not a number.
NIL

>
```

The reason we said "usually" is that this function can be made to behave like `error` by setting the value of the variable `*break-on-warnings*` to something other than `nil`.

Further change `convert`, which was redefined in Exercise 11.9.4, so that when the argument is given incorrectly, the system beeps, chooses an appropriate value, and continues the execution. In other words, give `convert` the ability of **error recovery**. Depending on the value of the argument, the system should choose the following values:

1. 10 if the argument was not a number;
2. 10 if the argument was a complex number;
3. 2 if the argument was smaller than 2;
4. the value rounded to the nearest whole number if the argument is bigger than 2 and smaller than 16.

You should start by defining an auxiliary function called `check-arg-to-convert` which checks the arguments to `convert`, beeps when necessary, and returns the appropriate choice above.

```
>(check-arg-to-convert 8)
8

>(check-arg-to-convert 'two)
Warning: TWO is not a real.
         I will use 10 instead.
10
```

```
>(check-arg-to-convert 0)
Warning: 0 is less than 2.
         I will use 2 instead.
2

>(check-arg-to-convert 100)
Warning: 100 is greater than 16.
         I will use 16 instead.
16

>(check-arg-to-convert 8/3)
Warning: 8/3 is not an integer.
         I will use 3 instead.
3
```

11.9.6 There is a system function called `get-decoded-time` which tells you the current date and time. This function does not take any arguments and it returns the following nine values:

1. seconds (0–59)
2. minutes (0–59)
3. hours (0–23)
4. day (1–31)
5. month (1–12)
6. year (the Christian Era, A.D.)
7. week (0–6, 0 is Monday, 1 is Tuesday, ..., 6 is Sunday.)
8. Daylight Savings Time or not (`t` or `nil`)
9. time zone (For example, California is -8.)

Using `get-decoded-time` and `format`, define a function `date` which displays the date as follows:

```
>(date)
Thu Jan 29 13:22:04 1987
NIL
```

11.9.7 Interpreter (5)
The function `read-eval-print-loop` has been redefined many times (see Exercise 3.5.3). Redefine it again using `format` so that it prints information more effectively. Also, change it so that it changes the prompt sign to * and displays the package name just before the prompt sign if the current package is other than `user`. To display a value, use `~&`, the format command which was introduced in Exercise 3.1.8.

```
>(read-eval-print-loop)

* (cons 1 2)
(1 . 2)

* (floor 8 5)
1
3

* (in-package 'tokyo)
#<"TOKYO" package>

TOKYO* (values 1 2)
1
2

TOKYO* (in-package 'user)
#<"USER" package>

* (good-bye 0)
Good-bye.
0

>
```

This is almost the top level loop in KCL.

Answers to Exercises

11.1.1 The values of these variables are different on different systems. The style of printing a stream may also be different. In KCL, the following thing happens:

```
>*terminal-io*
#<two-way stream 700406bc>

>*standard-output*
#<synonym stream to *TERMINAL-IO*>

>*standard-input*
#<synonym stream to *TERMINAL-IO*>
```

A `two-way stream` is a stream which can do both input and output. A `synonym stream` is a stream which forwards input or output to the stream that is the value of some variable. In KCL, both `*standard-input*` and `*standard-output*` forward input and output to the stream that is the value of `*terminal-io*` as shown above. Common Lisp system has four more predefined streams:

`*debug-io*`	input and output for debugging
`*error-output*`	output of error messages
`*trace-output*`	output of traces
`*query-io*`	input and output for user interaction

You can make the system do specialized things, for example, output only error messages to a file, by changing the value of these variables.

11.1.2 The author's KCL system duplicates each letter as follows. It even duplicates the prompt sign.

```
>(setq x *standard-output*
       (make-broadcast-stream
         *standard-output*
         *standard-output*))
#<broadcast stream 700406ae>

>(progn (setq *standard-output* x)
        nil)
NNIILL

>>(cons 'a 'b)
((AA  . BB))
```

```
>>"output"
""oouuttppuutt""
```

11.2.1

a. `(write '|Common Lisp| :escape nil)`
b. `(write #\a :escape nil)`
c. `(write '(#\a "What?") :escape nil)`
d. `(write 10 :base 2)`
e. `(write 10 :base 2 :radix t)`
f. `(write 10 :base 10 :radix t)`
g. `(write 'if :case :downcase)`
h. `(write 'if :case :capitalize)`
i. `(write '(a b c d e f) :length 3)`

11.2.2

```
123|123|
|123|123123
123Z#\Z
#\ZHey, You!"Hey, You!"
"Hey, You!"(A b c)(A #\b |c|)
(A #\b |c|)#(A b c)#(A #\b |c|)
#(A #\b |c|)
```

11.2.3

```
(defun write-strings (&rest strings)
  (cond ((endp (cdr strings))
         (princ (car strings)))
        ((endp (cddr strings))
         (princ (car strings))
         (princ " and ")
         (princ (cadr strings)))
        (t (do ((ss strings (cdr ss)))
               ((endp ss))
             (princ (car ss))
             (unless (endp (cdr ss))
               (if (endp (cddr ss))
                   (princ ", and ")
                   (princ ", "))))))
  (princ ".")
  (values))
```

The two special cases when there is one argument ("**and**" is not necessary) and when there are two arguments (the ", " is not necessary in front of the "**and**") are distinguished using a `cond` expression. When there are more than three arguments, a ", " is output between words except that ", **and** " is output between last two words. This is decided using an `if` form.

11.2.4

```
>(setq *print-base* 2)
[10]

>128
[10000000]

>(setq *print-radix* t)
T

>128
[#b10000000]

>(setq *print-base* 10)
[10.]

>128
[128.]
```

11.2.5

```
>x
[(NIL NIL NIL ...)]

>y
[((((NIL) NIL) (NIL) NIL) ((NIL) NIL) (NIL) ...)]

>(setq *print-level* 3)
3

>x
[(NIL NIL NIL ...)]

>y
[(((# NIL) (NIL) NIL) ((NIL) NIL) (NIL) ...)]
```

11.3.1

```
>'(#b10 #o10 #x10 #xf)
[(2 8 16 15)]

>(setq *read-base* 2)
2

>(list 10 (+ 10 100) (+ 10. 10) #o10)
[(2 6 12 8)]

>(setq *print-base* 2.)
[10]

>(cons (+ 100 10) (- 100 1))
[(110 . 11)]
```

11.3.2

```
(defun convert (i j)
  (let ((*read-base* i)
        (*print-base* j))
    (loop
      (princ "CONVERT>")
      (finish-output)
      (let ((x (read)))
        (unless (numberp x) (return x))
        (prin1 x)
        (terpri)
        (terpri)))))
```

11.4.1

```
(defun read-number ()
  (let ((n (do ((c (read-char) (read-char)))
               ((digit-char-p c)
                (digit-char-p c))
             )))
    (do ((c (read-char) (read-char)))
        ((not (digit-char-p c))
         (unread-char c))
      (setq n (+ (* n 10) (digit-char-p c))))
    n))
```

The first do expression skips characters which are not digits. If it finds a digit, it takes the corresponding numerical value as the initial value of n. The next do expression reads characters as long as it finds a digit and recomputes the value of n. It is important to back up over the nondigit character using unread-char. Otherwise, the system may not read the operator calculation symbols correctly.

```
(defun read-op ()
  (do ((c (read-char) (read-char)))
      ((member c '(#\+ #\- #\* #\/ #\.))
       c)
    ))
```

read-op is simpler and should simply read characters until it reads one of the designated five characters.

```
(defun calc ()
  (let ((n (read-number)))
    (do ((op (read-op) (read-op)))
        ((char= op #\.) n)
      (setq n
        (case op
          (#\+ (+ n (read-number)))
          (#\- (- n (read-number)))
          (#\* (* n (read-number)))
          (#\/ (round n (read-number))))))))
```

11.4.2

```
(defun print-queens (n column)
  (dolist (j (reverse column))
    (dotimes (k j) (write-char #\-))
    (write-char #\Q)
    (dotimes (k (- n j 1)) (write-char #\-))
    (terpri)))
```

11.5.1

```
(defvar *previous-trace-output* nil)
```

```
(defun begin-trace-to-file (file)
  (setq *previous-trace-output*
        *trace-output*)
  (setq *trace-output*
        (open file :direction :output)))


(defun end-trace-to-file ()
  (close *trace-output*)
  (setq *trace-output*
        *previous-trace-output*)
  (setq *previous-trace-output* nil))
```

The above function saves the value of `*trace-ouptut*` when tracing to the file starts and sets `*trace-output*` back to this value when tracing to the file is finished.

The above program is usable but does not work in unusual situations. For example, if the system executes `begin-trace-to-file` while tracing to a file, it may lose the value of `*trace-output*` which has been saved initially. We recommend that you improve this program so that it works in such unusual situations.

11.5.2

```
(with-open-file (out "symbols.out"
                     :direction :output)
  (do-external-symbols (s (find-package 'lisp))
    (print s out)
    (cond ((boundp s)
           (princ " = " out)
           (prin1 (symbol-value s) out))
          (t (princ " : unbound" out)))))
```

11.5.3 You can get the sum by evaluating

```
(with-open-file (in "numbers.in")
  (do ((n (read in nil nil)
          (read in nil nil))
       (sum 0 (+ sum n)))
      ((null n) sum)))
```

11.5.4

```
(defun copy-file (infile outfile)
  (with-open-file (in infile)
    (with-open-file (out outfile
                         :direction :output)
      (do ((c (read-char in nil 'eof)
              (read-char in nil 'eof)))
          ((eq c 'eof))
        (write-char c out)))))
```

We arranged for `read-char` to return the symbol `eof` when the system comes to the end of a file. However, any data structure other than a character could be returned at this time.

11.5.5

```
(defun count-lines (file)
  (with-open-file (in file)
    (let ((count 0)
          (last-char-was-newline-p t))
      (do ((c (read-char in nil 'eof)
              (read-char in nil 'eof)))
          ((eq c 'eof)
           (if last-char-was-newline-p
               count
               (1+ count)))
        (if (char= c #\Newline)
            (progn
              (incf count)
              (setq last-char-was-newline-p t))
            (setq last-char-was-newline-p nil)
            )))))
```

11.5.6 Let's use `copy-file` defined in Exercise 11.5.4.

```
(defun backup-file (original)
  (let ((backup (concatenate
                 'string
                 original
                 ".BAK")))
    (when (probe-file backup)
      (delete-file backup))
    (copy-file original backup)))
```

11.5.7

```
(defun show-file (file n i)
  (with-open-file (in file)
    (dotimes (line (1- n))
      (do ((c (read-char in nil 'eof)
              (read-char in nil 'eof)))
          ((eq c 'eof)
           (return-from show-file))
        (when (char= c #\Newline) (return))))
    (dotimes (line i)
      (do ((c (read-char in nil 'eof)
              (read-char in nil 'eof)))
          ((eq c 'eof)
           (return-from show-file))
        (write-char c)
        (when (char= c #\Newline) (return))))))
```

The first `dotimes` expression skips $n - 1$ lines. The `do` expression inside of this expression skips one line by repeatedly skipping one character until it finds the end of the line. If the system reaches the end of the file while doing this, the expression does nothing and ends the execution of `show-file`. The second `dotimes` prints on the terminal the next i lines. This `do` loop does the same thing as the previous `do` loop except it displays each character it reads.

11.6.1

```
(defun append-strings (&rest strings)
  (with-output-to-string (out)
    (dolist (string strings)
      (princ string out))))
```

11.6.2

```
(defun find-word (stream)
  (do ((c (read-char stream nil nil)
          (read-char stream nil nil)))
      ((or (null c) (alphanumericp c))
       (unless (null c) (unread-char c stream))
       c)
    ))
```

```
(defun read-word (stream)
  (do ((c (read-char stream nil nil)
          (read-char stream nil nil)))
      ((or (null c)
           (not (alphanumericp c)))
       (unless (null c)
         (unread-char c stream)))))

(defun count-words (stream)
  (do ((c (find-word stream)
          (find-word stream))
       (count 0 (1+ count)))
      ((null c) count)
    (read-word stream)))

(defun count-words-in-string (string)
  (with-input-from-string (stream string)
    (count-words stream)))
```

11.6.3

```
(defun count-words-in-file (file)
  (with-open-file (stream file)
    (count-words stream)))
```

11.7.1 First, define as follows:

```
>(defun |~-reader| (stream char)
   (list 'not (read stream)))
```

Then type

```
>(set-macro-character #\~ '|~-reader|)
```

The second argument could have been

```
#'|~-reader|
```

but, by specifying the function using a symbol, the system will use the function value of `|~-reader|` every time `read` sees a "~". Therefore, the user does not have to call `set-macro-character` each time the definition is changed.

11.7.2 In a system which uses ASCII codes,

```
>(dotimes (i #x80)
   (let ((char (code-char i)))
     (when (get-macro-character char)
       (print char))))
```

causes the system to display all the macro characters. KCL will show the following:

```
#\"
#\#
#\'
#\(
#\)
#\,
#\;
#\`
NIL
>
```

11.8.1 First, define as

```
>(defun |#v-reader| (stream char ignore)
   (concatenate 'vector (read stream)))
```

Then, type

```
>(set-dispatch-macro-character #\# #\v
   '|#v-reader|)
```

11.9.1

```
(defun display-ship (ship)
  (format t
    "~A at (~A,~A) with ~A units of power.~%"
    (ship-name ship)
    (ship-x ship)
    (ship-y ship)
    (ship-power ship)))
```

The format directive `~A` is used four times. All those except the first `~A` could be `~S`. If the first `~A` were to be written `~S`, the character " would be displayed and the whole output would look like

```
"USS Enterprise" at (1,2) with 97 units of power.
```

11.9.2

```
(with-open-file (out "symbols.out"
                     :direction :output)
  (do-external-symbols (s (find-package 'lisp))
    (if (boundp s)
        (format out "~%~S = ~S"
                s
                (symbol-value s))
        (format out "~%~S : unbound" s))))
```

11.9.3

```
(defun backup-file (original)
  (do ((i 0 (1+ i)))
      (nil)
    (let ((backup
           (format nil "~A.~A" original i)))
      (unless (probe-file backup)
        (copy-file original backup)
        (return)))))
```

11.9.4

```
(defun convert (i j)
  (cond ((not (integerp i))
         (error "~S is not an integer." i))
        ((not (<= 2 i 16))
         (error "~S is not between 2 and 16." i))
        ((not (integerp j))
         (error "~S is not an integer." j))
        ((not (<= 2 j 16))
         (error "~S is not between 2 and 16." j))
        (t
         (let ((*read-base* i)
               (*print-base* j))
           (loop
             (format t "~%CONVERT>")
             (let ((x (read)))
               (unless (numberp x) (return x))
               (format t "~S~%" x)))))))
```

11.9.5

```
(defun check-arg-to-convert (n)
  (cond ((or (not (numberp n))
             (complexp n))
         (warn "~S is not a real.~%~
                I will use 10 instead." n)
         10)
        ((< n 2)
         (warn "~S is less than 2.~%~
                I will use 2 instead." n)
         2)
        ((> n 16)
         (warn "~S is greater than 16.~%~
                I will use 16 instead." n)
         16)
        ((not (integerp n))
         (let ((x (round n)))
           (warn "~S is not an integer.~%~
                  I will use ~S instead." n x)
           x))
        (t n)))

(defun convert (i j)
  (let ((*read-base* (check-arg-to-convert i))
        (*print-base* (check-arg-to-convert j)))
    (loop
      (format t "~%CONVERT>")
      (let ((x (read)))
        (unless (numberp x) (return x))
        (format t "~S~%" x)))))
```

If the format character string is too long to fit on one line, it can continue to the next line by inserting ~ at the end of the first line as in the above definition. The system ignores the space after the ~ and the space at the beginning of the next line. In other words, the format character string

```
"~S is not a real.~%~
I will use 10 instead."
```

is equivalent to

```
"~S is not a real.~%I will use 10 instead."
```

11.9.6

```
(defun date ()
  (multiple-value-bind
      (sec min hour day month year week)
      (get-decoded-time)
    (format t "~A ~A ~D ~D:~D:~D ~D"
      (cdr (assoc week
                  '((0 . "Mon") (1 . "Tue")
                    (2 . "Wed") (3 . "Thu")
                    (4 . "Fri") (5 . "Sat")
                    (6 . "Sun"))))
      (cdr (assoc month
                  '((1 . "Jan") (2 . "Feb")
                    (3 . "Mar") (4 . "Apr")
                    (5 . "May") (6 . "Jun")
                    (7 . "Jul") (8 . "Aug")
                    (9 . "Sept") (10 . "Oct")
                    (11 . "Nov") (12 . "Dec"))))
          day hour min sec year)))
```

11.9.7

```
(defun read-eval-print-loop ()
  (unwind-protect
    (catch 'goodbye-bye-bye
      (let ((user-package (find-package 'USER)))
        (loop (if (eq *package* user-package)
                  (format t "~%* ")
                  (format t "~%~A* "
                    (package-name *package*)))
              (let ((form (read)))
                (if (null form)
                    (return)
                    (dolist
                        (value
                          (multiple-value-list
                            (eval form)))
                      (format t "~&~S~%" value)
                      ))))))
    (format t "~&Goodbye.~%")))
```

12 Programming

A Lisp program is prepared in a program file and is loaded to the system for the execution. In order to improve the efficiency of such a program, you should compile the program and load the object file created by the compiler.

`eval-when` and `compiler-let` control the compiler.

A big Lisp program should be divided into modules which can be organized using `provide` and `require`.

Properties of the Lisp system that can depend on the particular implementation of Common Lisp can be found by examining the variable `*features*`.

- summary of functions

  `(load` ⟨file name⟩ {`:verbose` `:print`}`)`
  `(ed` [⟨file name⟩]`)`

  `(time` ⟪expression⟫ `)`
  `(compile` ⟨function name⟩`)`
  `(compile-file` ⟨file name⟩ {`:output-file`}`)`

  `(eval-when` `(` ⟪when to evaluate⟫ ... ⟪when to evaluate⟫ `)`
    ⟪expression⟫ ... ⟪expression⟫ `)`

  `(compiler-let` `((` ⟪variable⟫ ⟨initial value⟩`)`
    ...
    `(` ⟪variable⟫ ⟨initial value⟩`)))`
    ⟪expression⟫ ... ⟪expression⟫ `)`

`(provide` ⟨module name⟩`)`
`(require` ⟨module name⟩ `[`⟨file name⟩`])`

`(describe` ⟨data⟩`)`
`(inspect` ⟨data⟩`)`
`(apropos` ⟨character string⟩ `[`⟨package⟩`])`

- summary of variables

  `*load-verbose*`
  `*features*`
  `*modules*`

12-1 A Program File

12.1.1 As described in Section 12.2 of *Introduction to Common Lisp*, the system function `load` can load object files as well as source files. Define the function `my-load` which can only load source files.

12-2 Compiler

12.2.1 Suppose you define a function which is used by `foo` after the function `foo` has been compiled. `foo` will still call this function correctly.

```
>(defun foo (x) (bar x))
FOO

>(compile 'foo)
FOO

>(defun bar (x) (1+ x))
BAR

>(foo 2)
3
```

However, as described in *Introduction to Common Lisp*, a macro which `foo` uses needs to be defined before the compilation of `foo`. Check how your system evaluates the last expression in the following interaction:

```
>(defun foo (x) (bar x))
FOO

>(compile 'foo)
FOO

>(defmacro bar (x) `(1+ ,x))
BAR

>(foo 2)
```

12.2.2 The special form `compiler-let` is used to control the expansion of a macro which refers to a global variable during compilation. Fill in [ ] with the appropriate values in the following interaction:

```
>(defvar *increment* 10)
*INCREMENT*

>(defmacro bar (x)
   `(+ ,x ,*increment*))
BAR

>(macroexpand '(bar x))
[ ]
T

>(let ((*increment* 100))
   (macroexpand '(bar x)))
[ ]
T

>(defun foo (x)
   (compiler-let ((*increment* 100))
     (bar x)))
FOO

>(foo 3)
[ ]

>(compile 'foo)
FOO

>(foo 3)
[ ]
```

12-3 Module Structure

12.3.1 Suppose there are four files, `a.lsp`, `b.lsp`, `c.lsp`, and `d.lsp`, which have the following content:

```
a.lsp:  (provide 'a)
        (defun a (x) (1+ x))

b.lsp:  (provide 'b)
        (require 'a "a.lsp")
        (defun b (x) (1+ (a x)))

c.lsp:  (provide 'c)
        (defun c (x) (1+ x))

d.lsp:  (provide 'd)
        (require 'b "b.lsp")
        (require 'c "c.lsp")
        (defun d (x) (1+ (c (b x))))
```

Fill in `[ ]` with appropriate values in the following interaction. Also, show which files will be loaded when you see "...".

```
>*modules*
NIL

>(load "b.lsp")
...

>*modules*
[ ]

>(load "b.lsp")
...

>*modules*
[ ]

>(load "d.lsp")
...

>*modules*
[ ]

>(d 0)
[ ]
```

12-4 Features

12.4.1 Fill in [ ] with appropriate values:

```
>*features*
(MV AOSVS COMMON KCL)

>'(#+mv 1
   #-mv 2
   #+vax 3
   #-vax 4)
[ ]

>'(a #+(and mv aosvs) b
     #+(and mv dgux) c
     #+(and vax unix) d
   e)
[ ]

>'(a #+(or mv aosvs) b
     #+(or mv dgux) c
     #+(or vax unix) d
   e)
[ ]

>'((x . 2)
   #+(or mv vax) (y . 3)
   #+(or sun news) (y . 4)
   #-(or mv vax sun news) (y . 0)
   (z . 10))
[ ]
```

12-5 Miscellaneous Information

12.5.1 Use `describe`, `inspect`, and `apropos` on your Common Lisp system. For example, try the following expression:

```
(describe 'cons)
(inspect '(a b))
(apropos "CONS" 'lisp)
```

Answers to Exercises

12.1.1

```
(defun my-load
    (file
     &key (print nil)
          (verbose *load-verbose*)
     &aux (eof (cons nil nil)))
  (with-open-file (in file)
    (when verbose
      (format t "~&Loading ~A~%" file))
    (do ((form (read in nil eof)
               (read in nil eof)))
        ((eq form eof))
      (if print
          (format t "~&~S~%" (eval form))
          (eval form)))
    (when verbose
      (format t "~&Finished loading ~A~%" file))
    t))
```

The format command `~&` outputs the rest of the characters on a new line. If `~&` occurs at the beginning of the new line, it does not do anything since it is already at the front of the new line. If not, it changes the line. This is a useful command to remember.

12.2.1 KCL will do the following:

```
>(foo 2)
Error: BAR is invalid as a function.
Error signalled by FOO.

Broken at FOO.  Type :H for Help.
>>
```

12.2.2

```
.......

>(macroexpand '(bar x))
[(+ X 10)]
T

>(let ((*increment* 100))
   (macroexpand '(bar x)))
[(+ X 100)]
T

.......

>(foo 3)
[103]

>(compile 'foo)
FOO

>(foo 3)
[103]
```

12.3.1

```
>*modules*
NIL

>(load "b.lsp")
Loading B.LSP
Loading A.LSP
Finished loading A.LSP
Finished loading B.LSP
T

>*modules*
[("A" "B")]

>(load "b.lsp")
Loading B.LSP
Finished loading B.LSP
T
```

```
>*modules*
[("A" "B")]

>(load "d.lsp")
Loading D.LSP
Loading C.LSP
Finished loading C.LSP
Finished loading D.LSP
T

>*modules*
[("C" "D" "A" "B")]

>(d 0)
[4]
```

12.4.1

```
>'(#+mv 1
   #-mv 2
   #+vax 3
   #-vax 4)
[(1 4)]

>'(a #+(and mv aosvs) b
     #+(and mv dgux) c
     #+(and vax unix) d
   e)
[(A B E)]

>'(a #+(or mv aosvs) b
     #+(or mv dgux) c
     #+(or vax unix) d
   e)
[(A B C E)]

>'((x . 2)
   #+(or mv vax) (y . 3)
   #+(or sun news) (y . 4)
   #-(or mv vax sun news) (y . 0)
   (z . 10))
[((X . 2) (Y . 3) (Z . 10))]
```

12.5.1 KCL on my system prints the following:

```
>(describe 'cons)
CONS - external symbol in LISP package
-------------------------------------------
CONS                                [Function]

Args: (x y)
Returns a new cons whose car and cdr are X
and Y, respectively.
-------------------------------------------
CONS                                    [Type]

A cons is a record structure containing two
components called the car and the cdr.
-------------------------------------------

>(inspect '(a b))
Type ? and a newline for help.
(A B) - cons
 nth 0:
    A ?

Inspect commands:
n (or N or Newline):inspects the field.
s (or S):           skips the field.
p (or P):           pretty-prints the field.
a (or A):           aborts the inspection.
u (or U) form:      updates the field.
e (or E) form:      evaluates the form.
q (or Q):           quits the inspection.
?:                  prints this.

 nth 0:
    A n
    A - internal symbol in USER package
 nth 1:
    B n
    B - internal symbol in USER package
 nthcdr 2:
    NIL n
    NIL - external symbol in LISP package
     value:
        NIL q
```

```
>(apropos "CONS" 'lisp)
CONSP  Function
CONS  Function
ACONS  Function
CONSTANTP  Function
DEFCONSTANT  Macro
```

Index

A

algebraic expression processing, 119
auxiliary function, 28

B

backup file, 199
bignum, 44
binary search tree, 74
branch, 74
broadcast stream, 191

D

data-driven, 127
declaration form, 157
derivative, 94
differentiation, 121
downward funarg problem, 102

E

Eight Queens, 150
error, 171
error recovery, 205
evaluation function, 122

F

Fibonacci array, 29
FILO(First-In, Last-Out), 150
fixnum, 44
form, 1
funarg problem, 102

G

game tree, 76

I

ignore declaration, 156
input and output for debugging, 208
input and output for user interaction, 208

L

lines in a file, 199

M

mini-max processing, 122

N

Newton's method, 171
next move, 76
node, 74
number of arguments, 91

O

output of error messages, 208
output of traces, 208

P

polynomial expressions, 94
propositional calculus, 125
propositional connectives, 125
propositional variable, 125

R

read-eval-print loop, 36
recursive data structure, 69
recursive definition, 69

S

self-tail-recursive, 29
shallow binding, 147
simplification, 121
situation, 76
sorting, 72
stone picking up game, 76
stream for both input and output, 208
symbol cells, 147

T

tail-recursively, 29
Tower of Hanoi, 31
tree structure, 74

U

unwinding, 149
upward funarg problem, 102

Carleton College Library
One North College Street
Northfield, MN 55057-4097

Taiichi Yuasa

Common Lisp Drill

Common Lisp Drill contains a wide variety of exercises for Common Lisp beginners, ranging from simple ones that ask for the values of expressions and fill in answers for empty parts of interactions to difficult ones that relate to the use of Lisp for applications in artificial intelligence. Each chapter includes a summary of the main ideas and a summary of the functions and variables. Special forms, macros, and functions are shown with their argument types, so the reader can easily refer to this section when using them to solve the problems or to program in Common Lisp. No assumptions are made about the reader's previous knowledge, and each new idea is explained within the exercise.

Common Lisp Drill is a collection of problems that have been used as exercises in the author's programming class for Seniors in the Department of Information Science, Kyoto University, and is perfectly suited as an independent tutorial for learning Common Lisp. It may be used as a companion volume to Taiichi Yuasa and Masami Hagiya's *Introduction to Common Lisp* to provide the reader with additional exercises and detailed solutions.

Also Available from Academic Press

Introduction to Common Lisp

Taiichi Yuasa and Masami Hagiya
Translated by Richard Weyhrauch and Yasuko Kitajima

Starting with a tutorial introduction to Lisp, the authors have carefully selected the most useful features of Common Lisp, enabling the reader to write programs even without referring to the Reference Manual. This textbook can be used in self study and is therefore also of interest to motivated programmers and Lisp experts interested in Common Lisp.

1987, 280 pp. Casebound/ISBN 0-12-774860-1

Cover Design: Hannus Design Associates

ISBN 0-12-774861-X